FASTER FILE MATCHING USING GPGPU'S

by

Deephan Venkatesh Mohan

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Summer 2010

© 2010 Deephan Venkatesh Mohan
All Rights Reserved

FASTER FILE MATCHING USING GPGPU'S

by

Deephan Venkatesh Mohan

Approved: ____

John Cavazos, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____

David Saunders, Ph.D.
Professor and Chair of the Department of Computer and Information Sciences

Approved: ____

George H.Watson, Ph.D.
Dean of the College of Arts and Sciences

Approved: ____

Debra Hess Norris, M.S.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr.John Cavazos, for giving me the impetus to work on this exciting project. His timely advice and suggestions were the cornerstones for the completion of this thesis. I would also like to thank my friends(arul, vinu, chetan, naran, nimmy) for the much needed encouragement during tough times.

I dedicate this work to my mom, dad and sister for their eternal love and affection which keeps me going all the time, my uncle and aunt for their innocent words of wisdom.

TABLE OF CONTENTS

LIST OF FIGURES . . . . **viii**
LIST OF TABLES . . . . **x**
ABSTRACT . . . . **xi**

Chapter

1 INTRODUCTION **1**

1.1 Motivation 1
1.2 Problem 2
1.3 Thesis Contribution 2
1.4 Thesis Organization 2

2 BACKGROUND AND RELATED WORK **4**

2.1 Introduction to GPU computing 4

2.1.1 Objective 4
2.1.2 Applications 5

2.2 Introduction to File Matching 5

2.2.1 Objective 6
2.2.2 Applications 6

2.3 Related Work 6

2.3.1 GPU Acceleration of Security Algorithms 7
2.3.2 Parallel Implementation of Dedicated Hash Functions 8

3 COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA) . . 9

3.1 Introduction 9
3.2 The CUDA Programming Model 9

3.2.1 CUDA Thread Hierarchy 10
3.2.2 CUDA Memory Hierarchy 10

3.3 Structure of a CUDA program 12

3.3.1 Host component 12
3.3.2 Device component 12

3.4 Layout of a CUDA application 13

4 THE MESSAGE-DIGEST 6 ALGORITHM (MD6) 14

4.1 Introduction 14
4.2 MD6 Design 14

4.2.1 Merkle tree 15
4.2.2 MD6 Inputs 16
4.2.3 MD6 Compression Function 16

4.3 Analysis of Sequential MD6 Implementation 17

5 DESIGN AND IMPLEMENTATION OF CUDA MD6 20

5.1 Framework of CUDA MD6 20

5.1.1 Design of the Host component 20

5.1.2 Design of the Device component 21

5.2 Recursive Vs Iterative MD6 22

5.2.1 Recursive MD6 22

5.2.1.1 Pseudocode for the Recursive version of MD6 22

5.2.2 Iterative MD6 24

5.2.2.1 Pseudocode for the Iterative version of MD6 25

5.3 CUDA implementation of MD6 26

5.3.1 Description of Host Function 26
5.3.2 Description of Kernel Function 26
5.3.3 Pseudocode for the CUDA implementation of MD6 27

5.4 CUDA MD6 for file matching 28

6 EVALUATION OF CUDA MD6 30

6.1 Experiment 1: Executing CUDA MD6 on single files 30

6.1.1 Analysis of Experiment 1 30

6.2 Experiment 2: Executing CUDA MD6 on archive of files 31

6.2.1 Analysis of Experiment 2 31

6.3 Experiment 3: Executing CUDA MD6 with varying MD6 Buffer size . 32

6.3.1 Analysis of Experiment 3 33

6.4 General Analysis 33

7 CONCLUSION AND FUTURE WORK 36

7.1 Conclusion 36
7.2 Future work 37

BIBLIOGRAPHY **38**

Appendix

A APPENDIX **43**

A.1 Host component 43
A.2 Kernel component 43

LIST OF FIGURES

3.1 The CUDA Memory Hierarchy 11

3.2 Layout of a CUDA program 13

4.1 Structure of a Merkle tree: The diagram shows a Merkle tree with 2 levels. The computation proceeds from the bottom level to the top. Each node represents a data chunk. The hash computation of a node takes place when the sub-hashes of all the child nodes have been computed. If the number of child nodes is less than 4, then the rest of the hash is padded with zeroes. This example presents a Merkle tree for a data of 7KB (assuming the buffer size to be 512 Bytes). . 15

4.2 Description of MD6 compression function 18

4.3 Snapshot of GProf profile results for the original MD6 implementation 19

5.1 Hash storing mechanism in Recursive MD6: In this example, MD6 sub-hash computation for the 13th node is in progression. The status of the stack is shown. Each highlighted tile in the stack represents a sub-hash of size - 16 MD6 words. At this point, 3 nodes at level 1 have their sub-hash inputs. The final hash will be computed after the 4th node at level 1 receives it sub-hash input. . 23

5.2 Hash storing mechanism in Iterative MD6 24

6.1 CUDA speedup experiment on individual files (Direct Hashing): The program was executed on individual files of sizes ranging from 25 MB to 1GB with the MD6 Buffer size preset to 512 bytes and the number of compression rounds = 72. 31

6.2 CUDA speedup experiment on archive of files (Recursive Hashing): The program was executed individually each time on an archive of files of sizes ranging from 100 MB to 8GB with the MD6 Buffer size preset to 512 bytes. 32

6.3 Real time speedup of CUDA MD6 with varying MD6 buffer sizes: CUDA MD6 was executed with varying CUDA buffer sizes of 512 Bytes, 1KB, 1.5KB and 2KB on a 100MB source file. 33

6.4 CUDA Speedup comparison with and without CUDA overhead (Graph): The graph shows the speedup comparison with and without CUDA overhead for the direct hashing experiment. 34

6.5 CUDA Speedup with different number of compression sounds: The graph shows the speedup comparison with varying number of compression rounds on different file sizes. 35

A.1 Host component: md6sum.cpp 44

A.2 Kernel component: md6_kernel.cu 52

LIST OF TABLES

6.1 CUDA Speedup comparison with and without CUDA overhead for the direct hashing experiment 34

ABSTRACT

File matching is an important topic in field of forensics and information security. With the increasing popularity of GPU computing for scientific research and other commercial purposes, there is a desire to solve problems in a faster and low-cost effective manner. There is a need for the design of faster and effective parallelizable algorithms to exploit the parallelism offered by the multi-core GPUs. One particular application that could potentially benefit from the massive amount of parallelism offered by GPUs is file matching. File matching involves identifying similar files or partially similar files using file signatures (I.e., hashes). It is a computationally expensive task, although it provides scope for parallelism and is therefore well suited for the GPU. We address the problem of faster file matching by identifying the parallel algorithm that is best suited to take advantage of GPU computing.

MD6 is a cryptographic hash function that is tree-based, highly parallelizable, and can be used to construct the hashes used in file matching applications. The message M to be hashed can be computed at different starting points and their results can be aggregated as the final step. We implemented a parallel version of MD6 on the GPUs using CUDA by effectively partitioning and parallelizing the algorithm. To demonstrate the performance of **CUDA MD6**, we performed various experiments with inputs of different sizes and varying input parameters. We believe that **CUDA MD6** is one of the fast and effective solutions for identifying similar files that are currently available.

Chapter 1

INTRODUCTION

This chapter presents a brief insight into parallel computing and how it can be exploited to solve the problem of file matching. As a first step, the problem is presented, the approach and contribution of this thesis is enumerated and the organization of the thesis is outlined in the last section.

1.1 Motivation

In the era of multi-chip / multi-core processors, there has been an increasing interest for parallel computing paradigms and parallel algorithms. The primary objective of designing parallel algorithms is to solve common problems in a fast and efficient manner by utilizing existing parallel computing architectures [1].

GPU computing is the newest trend in parallel computing enabling users to run their applications in a cost-effective faster environment. With the advent of **CUDA** programming, it is now possible to parallelize applications on General Purpose Graphical Processing Unit (GPGPU). The major advantage of GPU computing is that the computational complexity in parallelizing applications is highly minimized compared to other parallel programming paradigms. **GPGPUs** are now deployed widely in various fields like medicine, imaging, research and academia for executing computationally intensive operations. The computational power of **GPGPUs** has been exploited in various fields of interest [2]. An outline about GPU computing and its applications is given in Section 2.1.

1.2 Problem

File matching is an important problem and is fundamental in the field of forensics. The problem chosen for parallelization on the **GPGPUs** in this work is the **MD6** algorithm, which is the newest in the "*message digest*" (MD) series of algorithms, generally deployed to find checksum or file hashes. The output produced by these algorithms ensure that there is no feasible way to determine the input. Section 2.2 provides a brief literature on the related work that has been done on the topic of file matching, its applications and explores some of the tools that have been deployed for file matching.

1.3 Thesis Contribution

The focus of this work is based on parallelizing **MD6** on the **GPGPUs** using the **CUDA** programming model. The **MD6** algorithm was ported to the **GPGPUs** and it was further deployed to solve the problem of file matching. We call our implementation **CUDA MD6** which is our solution for faster file matching to identify similar files on a large scale.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 explores the topics of GPU computing and file matching, their objective and applications. The final section discusses the related work that has been done on file matching.

Chapter 3 provides a brief overview of the CUDA programming model, the memory hierarchy of GPGPUs and the general structure of a CUDA application.

Chapter 4 explores the internal design of the MD6 Algorithm and it ends with the call graph analysis of the original implementation.

Chapter 5 gives a detailed insight into the design of CUDA MD6 and its application to file matching.

Chapter 6 discusses the various experiments done using CUDA MD6. The final section provides a general discussion on improving the GPU performance.

Chapter 7 presents the conclusion and the description of the future work that remains to be done.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter explores the topics of GPU computing and file matching, their objectives and applications. The final section discusses the related work that has been done on file matching.

2.1 Introduction to GPU computing

GPU computing is the latest parallel programming paradigm that has been gaining popularity, since it allows the users to effectively map their applications to the GPGPUs which are powerful inexpensive devices that can perform massive amount of parallel computation and can be integrated into commodity desktop or laptop computers. This is made possible due to the presence of multi-core GPGPUs which facilitates independent computation, thereby cutting the computational and communication complexity involved in parallelizing the application across a cluster.

2.1.1 Objective

Each computing core in the GPGPUs can carry out an independent operation, thereby providing the scope for massive parallelism. The crux of GPU computing resides on the *Single Instruction Multiple Data* (**SIMD**) model. Each computing core in the GPGPU's engage light-weight threads [3] to carry out the computation. There are various fundamental computational models for both parallel and serial computation [4].

The two primary GPU programming models popular today are

- **CUDA** programming model
- **Brook++** programming model

Brook++ is the primary parallel programming platform adopted for parallel programming on the AMD architecture using ATI GPUs. Parallelization is facilitated by making use of the ATI Stream SDK libraries. This work however, is based on the **CUDA** programming model on the Intel architecture using NVIDIA GPUs. Parallelization is facilitated by making use of the libraries in the CUDA SDK.

2.1.2 Applications

GPU computing has found ubiquitous usage in most fields of interests where there is scope for parallel computing and the necessity to perform computationally intensive tasks. For example, GPU computing has found pervasive use in the fields like climate research, computational chemistry and biology, engineering analysis, financial analysis, genetic research, bioinformatics, physics, rendering, security, seismic processing, signal processing, simulations, video transcoding. Many fields and their corresponding research is driven by the CUDA programming model [2] and the OpenCL programming model for AMD platform[5].

2.2 Introduction to File Matching

File matching is an important task in the field of forensics and information security. Every file matching application is driven by employing a particular hash generating algorithm. A hash generating algorithm is used for checksum calculations. The crux of the file matching application relies on the robustness and integrity of the hash generating algorithm.

2.2.1 Objective

File matching in general can be used to identify similar files or partially similar files using file signatures or hashes, generated by making use of various hash generation algorithms. A hash makes use of a cryptographic hash function which is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, refer Section 2.3. File matching is based on the simple logic of hash-comparison generated by these algorithms. The nature of file-matching however, is objective to the kind of hash generation algorithm. Some of the popular hash generating algorithms used in these applications are checksum generation algorithms like MD5[6], SHA-1[7], SHA-256[8], Tiger[9], Whirlpool[10], RSA[11], rolling hash[23].

2.2.2 Applications

File matching finds wide usage in anti-virus programs, system optimization tools and integrity checking programs. Depending on the nature of the application, the hash generating algorithm is selected. This is because each cryptographic algorithm has its own threshold for security and most of them are vulnerable to decoding attacks. For example, vulnerable algorithms like MD5 are used for general tasks like integrity checking, since there is no necessity for information security in this work. On the contrary, highly stable algorithms like RSA are resistant to timing attacks and thus used in message authentication. A timing attack is a reverse engineering procedure of decoding the message given the hash.

2.3 Related Work

File matching applications like the package **HASHDEEP**[12] makes use of message digest algorithms like MD5[6], SHA-1[7], SHA-256[8], Tiger[9] and Whirlpool[10] hashing algorithms to compute similarity of files. The package **SSDEEP**[13] uses context-triggered-piecewise hashing. CTPH[14] makes use of a sophisticated hash

matching algorithm for identifying partial file matches. The spamsum algorithm is used for identifying partial file matches. SSDEEP was adapted from the spam email detector called spamsum[23]. Spamsum[23] can identify messages that are similar but not identical to samples of known spam. The spamsum[23] was in turn based on the rsync checksum[24] developed by the same author. FTimes[25] is a free open source system baselining and evidence collection tool. FTimes[25] essentially takes a list of directories and files as input, iteratively and/or recursively gathers attributes and returns a set of text records containing one or more delimited attributes for each object encountered, uses message digest algorithms like MD5, SHA-1. Extreme GPU Bruteforcer, PasswordsPro are professional GPU password recovery programs for MD4, MD5, SHA-1, SHA-256 hashes. Various open source forensic tools [48] use these hashing functions for utilities like data acquisition, password recovery, application analysis.

2.3.1 GPU Acceleration of Security Algorithms

The NIST proposal for MD6 [20] discusses the GPU implementation. An efficient implementation for MD5-RC4 encryption using NVIDIA's CUDA programming framework [26] showing that the GPU-based implementation exhibits a performance gain of about 3-5 times speedup for the MD5-RC4 encryption algorithm. The MD5, SHA implementation on the GPUs [46] demonstrate the potential for various GPU optimizations that can be performed. The MD5 algorithm parallelized on the GPUs is used as a password recovery system [43]. The parallel version of SHA1/MD5/MD4 for ATI & nVidia GPUs [44] is used as a hash cracker. The tiger hash function implementation on GPUs [47] is used for random number generation. Apart from these implementations, there are numerous other applications using these hashing functions implemented on the GPUs for various other utilities other than file matching.

2.3.2 Parallel Implementation of Dedicated Hash Functions

The NIST proposal for MD6 [19] discusses the hardware implementation of MD6 on *Field Programmable Gate Arrays*(FPGA) (Section 5.2) and the ASIC implementation (Section 5.3). The various hardware design tradeoffs of MD6 are discussed in Section 5.1. Various algorithms like MD5 [27, 28, 29, 30], SHA-1[37], SHA-2[38], SHA-256 [31, 33], Whirlpool[39, 40] and SHA-512 [33] have been implemented on the FPGAs. The parallel implementations of the MD4-family of hash functions (MD4, MD5, SHA-1, RIPEMD-160) have been parallelized for the 32-bit Intel Pentium superscalar processors [32]. The extensive software performance analysis of targeted hash functions like MD5, RIPEMD-128 -160, SHA-1 -256 -512 and Whirlpool on Pentium III utilizing MMX instructions [34] have been experimented for two and three block parallelization. The hardware implementation of SHA-2 (of the same SHA standard) [35] contrasts the performance of the parallel implementation with that of the unrolled version of the algorithm. The hardware implementation of whirlpool function [36] discusses the various optimizations based on throughput. Several studies [41, 42] have examined the feasibility of parallel hash crackers, but nearly every system was a classic HPC application built using the Message Passing Interface (MPI). The parallel implementation of MD5 implemented on PS3 and CPU (Core 2 Duo, 2x quad Xeon, Pentium D 2.8 and AMD X2) [43] is used as a password recovery system.

Chapter 3

COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

This chapter provides a brief overview of the CUDA programming model, the memory hierarchy of GPGPU's and the general structure of a CUDA application.

3.1 Introduction

Compute **U**nified **D**evice **A**rchitecture **(CUDA)** is the primary computing engine in NVIDIA graphic processing units. The software architecture of CUDA provides a parallel programming model which allows programmers to partition the problem into coarse sub-problems that can be solved independently in parallel. The programmer can write CUDA applications in high-level languages like C, C++, Fortran.

3.2 The CUDA Programming Model

In the CUDA programming model, the GPUs act as the primary computing device. The GPU functions as a secondary processor to the main CPU or host. The computationally intensive tasks are transferred to the GPU device. The portion of the program that performs the same function but operate on different data can be isolated into a *kernel*. The amount of parallelism involved in these tasks can be exploited by the GPU threads. A good description of the device component is given in Section 3.3.2. The device component or the kernel is invoked from the CPU side

of the program called a host component. The host component is discussed in detail in Section 3.3.1.

3.2.1 CUDA Thread Hierarchy

The thread hierarchy is useful in deploying the kernel component of the CUDA application. The following terms form an integral part of any CUDA kernel component.

Grid of Thread blocks: A grid encompasses a limited number of thread blocks. Each block is identified by its block ID, which is the block number within the grid. The coordinates of each grid is two dimensional in nature specified by (D_x, D_y). The maximum size of each dimension of a grid of thread blocks is 65535.

Thread block: A thread block is a batch of threads that are deployed at the same time to efficiently share data through shared memory and they execute in synchronism to coordinate memory accesses. Each thread is identified by its thread ID, which is the thread number within the block. The coordinates of the threads can be either two- or three-dimensional. The maximum number of threads per block is 512. The maximum sizes of the x-, y-, and z-dimension of a thread blocks are 512, 512, and 64, respectively. CUDA threads may access data from multiple memory spaces during their execution. The global, constant, and texture memory spaces are optimized for different memory usages and they are briefly addressed in the next section.

3.2.2 CUDA Memory Hierarchy

The GPU device is implemented as a set of multiprocessors as illustrated in Figure 3.1. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD).

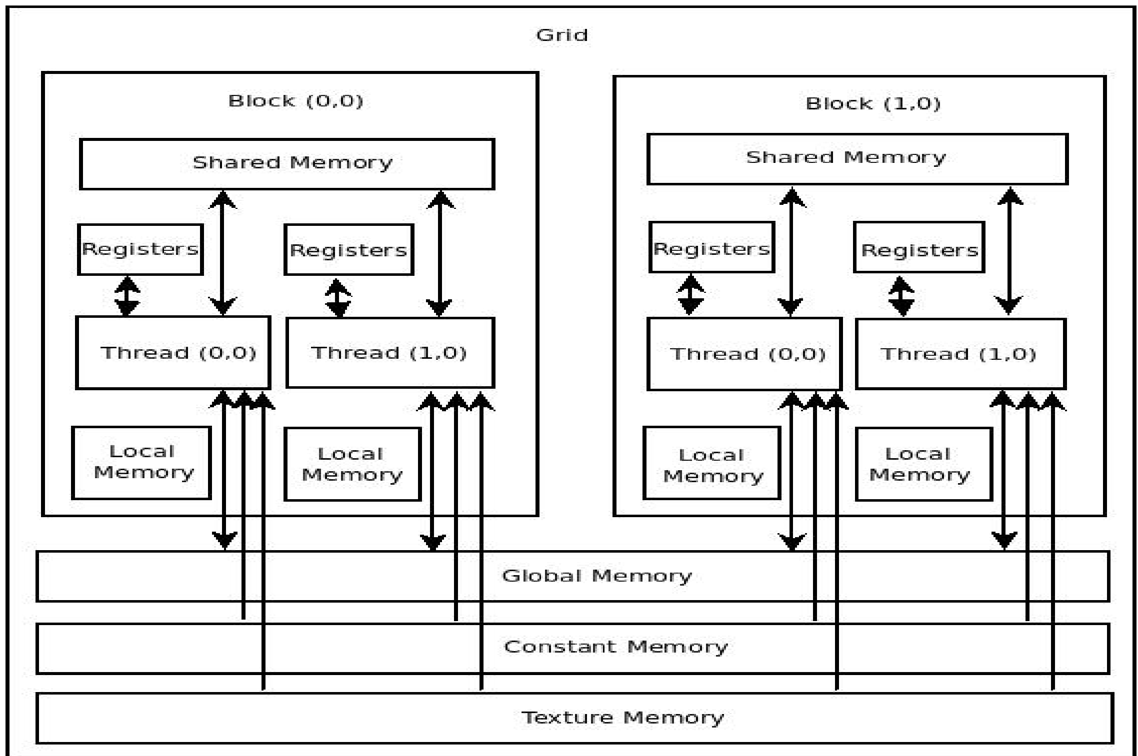

Figure 3.1: The CUDA Memory Hierarchy

As per the description in the CUDA programming guide [15], each CUDA enabled GPU multiprocessor has on-chip memory of the four following types:

1. One set of local 32-bit *registers* per processor, 2. A *parallel data cache or shared memory* that is shared by all the processors and implements the shared memory space, 3. A *read-only constant cache* that is shared by all the processors, 4. A *read-only texture cache* that is shared by all the processors.

The texture memory cache is faster than other types of memory due to higher bandwidth. Each CUDA supported GPU can have more than one multiprocessor. The texture memory is only 8KB per multiprocessor. The number of registers

per multiprocessor is 8192. The total shared memory space of 16KB is organized into 16 banks. The amount of constant memory available is 64 KB (i.e 8 KB per multiprocessor).

3.3 Structure of a CUDA program

As inferred in Section 3.2, a typical CUDA program consists of two main components:

- Host component
- Device component

They are illustrated in the following sections.

3.3.1 Host component

A host component runs on the host(CPU) and provides functions to control and access one or more GPUs from the host. The host component is typically responsible for initiating memory allocation /deallocation on GPU's, the transfer of data between the host and device and vice versa. These helper functions that serve as an interface between the host and device are generally declared using the __**global**__ qualifier in the kernel. An example of the host component is given in Section 5.3.1.

3.3.2 Device component

The device component is typically the kernel function that performs the parallelization on the GPUs. They are initiated by the call from the host component through the __**global**__ function in the kernel. The __**global**__ functions which are callable from both host and device code, execute only on the device. The kernel can essentially contain calls to other functions declared as __**global**__ or __**device**__. A __**device**__ function runs on the device and provides device-specific functions. A

__**device**__ function therefore cannot be invoked on the host code. An example of the device component is given in Section 5.3.2. Using __**global**__ or __**device**__ qualifiers usually imposes some programming restrictions [15].

Figure 3.2 gives the overall layout of a typical CUDA application. As discussed in Sections 3.2.1 and 3.3, it gives a clear example of how a typical CUDA program makes use of the CUDA programming model.

3.4 Layout of a CUDA application

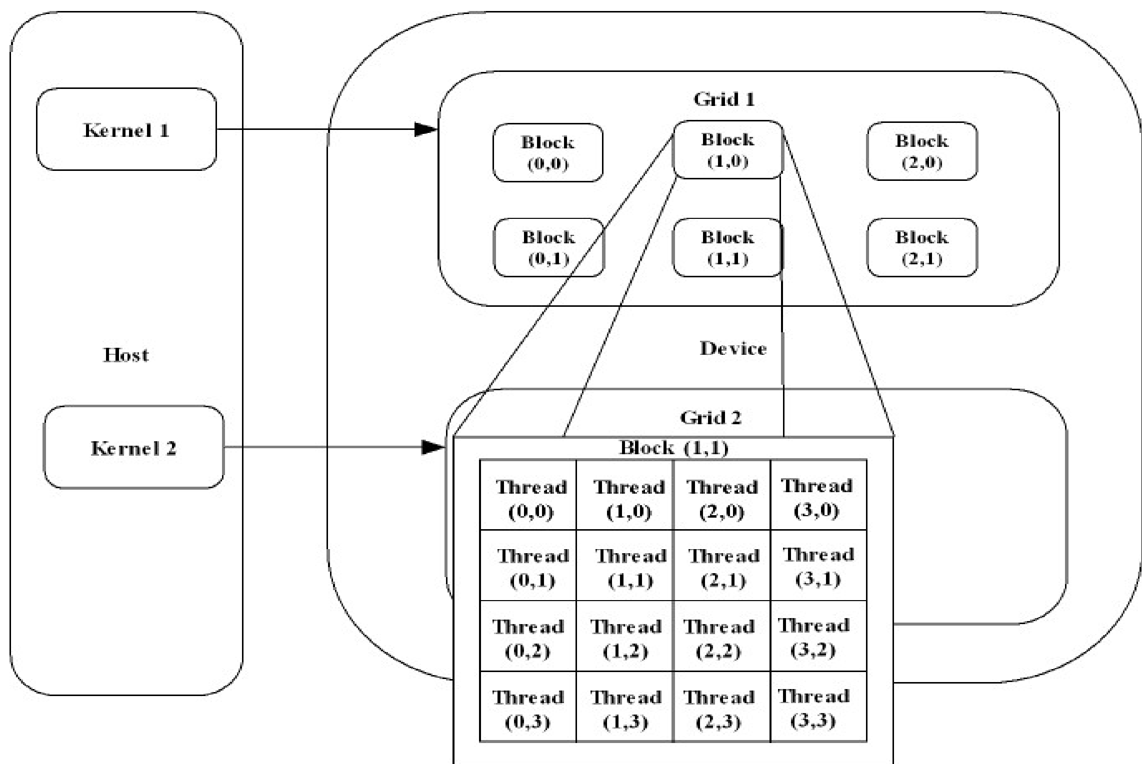

Figure 3.2: Layout of a CUDA program

Chapter 4

THE MESSAGE-DIGEST 6 ALGORITHM (MD6)

This chapter explores the working of the MD6 algorithm and ends with the call graph analysis of the original implementation.

4.1 Introduction

MD6[19] is the latest of the "message digest" algorithm and is ideally suited to exploit the parallelism presented by today's GPGPU architectures. The brunt of computation takes place in the MD6 compression function. The MD6 compression functions are data parallel and therefore multiple compression functions can be run in parallel on the processing units of the GPGPU, thereby achieving high GPU utilization.

4.2 MD6 Design

The design of MD6 is based on tree structure, it is highly parallelizable. Graphically, the structure of MD6 can be represented as quartenary trees of height at most L (L refers to the number of levels in the tree), each containing 4^L leaf chunks. The leaves of the quartenary tree represent the data and the final hash value is computed at the root. The sub-hash at each node except the leaves are computed by combining the values produced at the children of a particular node. In the parallel implementation of MD6, L "parallel" passes can be made over the data, performing a four-fold reduction at each level. Each compression function can

be further be parallelized due to the imminent parallel design. The default size of a MD6 word is 8 Bytes. Typically, the MD6 Buffer size is 64 Bytes (8 MD6 words).

4.2.1 Merkle tree

The design of MD6 is based on a highly parallelizable data-structure known as **"Merkle tree"**. Merkle trees are especially used in hashing function to compute parallel hashes. A tree-based design is the most natural approach for exploiting parallelism in a hash function. The computation proceeds from the leaves towards the root, with each tree node corresponding to one compression function computation. Each call to the MD6 compression function takes care of computing the sub-hash which corresponds to a node in the Merkle tree. Each level in the Merkle tree corresponds to a compression level with the root being the final hash computed by the algorithm. In the MD6 implementation, the MD6 buffer size determines the number of levels in the tree. This parameter also determines the number of leaves and the bounds for parallelism. An example of a Merkle tree is shown in Figure 4.1.

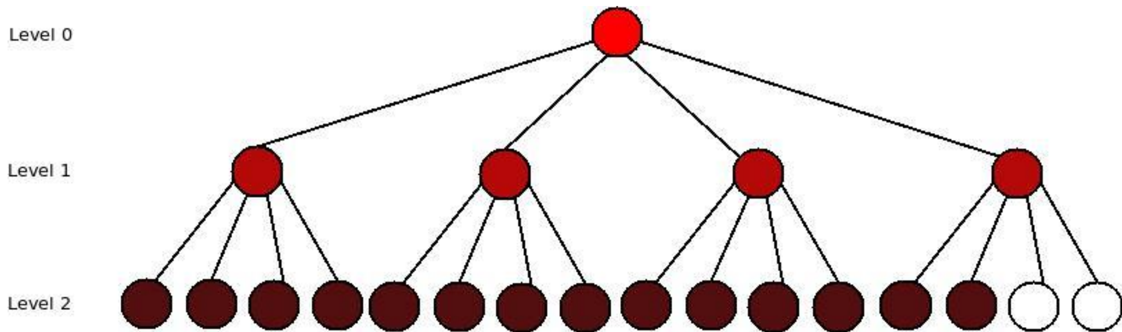

Figure 4.1: Structure of a Merkle tree: The diagram shows a Merkle tree with 2 levels. The computation proceeds from the bottom level to the top. Each node represents a data chunk. The hash computation of a node takes place when the sub-hashes of all the child nodes have been computed. If the number of child nodes is less than 4, then the rest of the hash is padded with zeroes. This example presents a Merkle tree for a data of 7KB (assuming the buffer size to be 512 Bytes).

4.2.2 MD6 Inputs

As described in the MD6 proposal [19], the MD6 hash function takes two mandatory inputs, M and d, where M is the message to be hashed and d is the desired digest length of the final hash. When the message M is available initially, the hashing operations can be initiated at different starting points within the message and their results can be aggregated as the final step. The value d must be known at the beginning of the hash computation, as it determines the length of the final MD6 output. The desired length of d is between 0 and 512. The MD6 hash function can also take two optional parameters, Key K and r. Key K is commonly used for security purposes. Round r is a parameter that controls the number of times the MD6 compression function can be invoked on a given MD6 chunk.

4.2.3 MD6 Compression Function

The MD6 compression function takes an input of fixed length and produces a data of shorter length. The default size of MD6 buffer is 64 words. A transformation from 64-word to 89-word input is the preprocessing step done as a part of the compression routine. The 89-word input to the compression function f contains a 15-word constant Q, an 8-word key K, a unique ID word U , a control word V and a 64-word data block B. K is a nil set, since this work does not concern security. B is the actual MD6 Buffer which contains the data. The default size of this data block is 512 Bytes. U and V are auxillary inputs that are part of the compression mechanism. The NIST proposal for MD6 [19] gives a detail description of what these parameters mean and how the compression function works. The description of MD6 compression function as quoted in [19] is given in Figure 4.2.

The compression function takes the **feedback tap positions** t_0, t_1, t_2, t_3, t_4, each in the range 1 to n-1 = 88, as parameters. **Feedback tap positions** are specific data units in the current CUDA buffer that are used as feedback parameters in the compression function. The MD6 compression function achieves a fourfold reduction

in size from data block to output - four chunks fit exactly into one data block, and one chunk is output. The final hash value generated is exactly d bits in length.

4.3 Analysis of Sequential MD6 Implementation

The sequential MD6 program was compiled and linked with profiling enabled and was subsequently executed to generate the profile data file. The GNU Profiler was then run on the profile data to determine the portions of the program taking most of the execution time. It was found that the function md6 main compression loop took 96.62% of the execution time (refer to Figure 4.3). This is the same function that performs the MD6 compression.

To test the possibility of parallelizing the function md6 main compression loop, a detailed dependency analysis was performed on the data block which is used by the function. It was found there was potential to parallelize each compression level and rounds of the algorithm. Each level computes the MD6 sub-hash for the given input. Each round of this compression perform 16 independent operations on each data block.

Description of MD6 compression function:

Input:

N : An input array $N[0..n-1]$ of n = 89 words.
r : A non-negative number of rounds.

Output:

C: An output array $C[0..c-1]$ of length c = 16 words.

Parameters:

$c, r, t_0, t_1, t_2, t_3, t_4, r_i, l_i, S_i$ where $1 \leq t_i \leq n$ for all $0 \leq i \leq 4$, $0 < r_i, l_i \leq w/2$ for all i, and S_i is a w-bit word for each i, $0 \leq i < t$.

Procedure:

Let t = rc. (Each round has c = 16 steps.)
Let $A[0..t+n-1]$ be an array of $t+n$ words.

Initialization:
$A[0..n-1]$ is initialized with the input $N[0..n-1]$.

Main computation loop:
for $i = n$ to $t+n-1$: /* t steps */
$x = S_{i-n} \oplus A_{i-n} \oplus A_{i-t_0}$
$x = x \oplus (A_{i-t_1} \wedge A_{i-t_2}) \oplus (A_{i-t_3} \wedge A_{i-t_4})$
$x = x \oplus (x \gg r_{i-n})$
$x = x \oplus (x \ll r_{i-n})$

Truncation and Output:
$A[t+n-c..t+n-1]$ as the output array $C[0..c-1]$ of length c = 16.

Figure 4.2: Description of MD6 compression function

Snapshot of GProf Results:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
96.62	40.94	40.94	956417	0.00	0.00	md6_main_compression_loop
2.190	41.87	0.930	45907920	0.00	0.00	md6_byte_reverse
0.610	42.13	0.260	717312	0.00	0.00	md6_reverse_little_endian
0.300	42.26	0.120	956417	0.00	0.00	md6_pack
0.070	42.28	0.030	717311	0.00	0.00	md6_process
0.050	42.30	0.020	956417	0.00	0.00	md6_compress_block
0.050	42.33	0.020	956417	0.00	0.00	md6_make_nodeID
0.050	42.34	0.020	358656	0.00	0.00	md6_update

Figure 4.3: Snapshot of GProf profile results for the original MD6 implementation

Chapter 5

DESIGN AND IMPLEMENTATION OF CUDA MD6

This chapter provides the technical specifications for running CUDA applications and the design description of CUDA MD6 package.

5.1 Framework of CUDA MD6

The original implementation of MD6 is not applicable for direct porting to CUDA. This is because the original implementation was written as a stack-based recursive version suited for serial implementation, refer Section 5.2.1. Thus, its necessary to redesign this function before we can parallelize with CUDA. The following gives the outline design of the major components of the CUDA MD6 package.

5.1.1 Design of the Host component

The host component is directly responsible for handling the data of CUDA MD6. This includes the following tasks strictly in the same order:

- Reading the source file
- Memory allocation on the GPU device
- Data transfer between host and device
- Invoking the kernel
- Data transfer between device and host

- Memory deallocation on the GPU device

- Store / Compare final hash

The host component invokes a **__global__** function which is a kernel that will be executed on the GPU device. A call to the kernel function must specify the execution configuration for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the kernel on the GPU. It is specified by inserting an expression of the form $<<< D_g, D_b, N_s >>>$ where

1. D_g is a three dimensional data type (dim3) and specifies the dimension and size of the grid. 2. D_b is of type dim3 and specifies the dimension and size of each block. 3. N_s is an optional parameter of type size_t and specifies the number of bytes in shared memory that is designated for use by each block.

dim3 and size_t are data types defined in the CUDA API , refer to CUDA Programming guide [15].

5.1.2 Design of the Device component

The kernel contains the functions to perform the MD6 compression in a thread-based design. As stated in Section 4.2.3, each of the 16 steps of the compression function can be executed by independent CUDA threads. Each kernel function has limited local memory access where local state variables can be defined. Common data elements can be stored in the shared memory for faster memory access. Shared variables are defined using the **__shared__** qualifier. Global memory access although slow when compared to other memory schemes, has both read-write privileges and adequate memory storage. The size of global memory is limited by the physical space available on the GPU device.

5.2 Recursive Vs Iterative MD6

The Merkle tree structure and the compression function provides the scope for the MD6 Algorithm to be parallelized effectively on the GPGPUs. The original version of MD6 was however written as a stack based recursive version making it unsuitable to be parallelized. The algorithm was thus rewritten as an iterative algorithm to make suitable to be parallelized. This section will provide the pseudocode of the recursive implementation of MD6 and a discussion of its unsuitability for parallelizing it for the GPU. The last section gives the description of the iterative version of MD6 algorithm which was rewritten to make it suitable to the CUDA architecture.

5.2.1 Recursive MD6

The original algorithm was written as a stack-based algorithm for efficient memory usage. It dealt with the compression of only one MD6 buffer chunk at a time and was therefore implemented as a serial program. This version did not support parallelism. In the recursive approach, the hash values for each level was stored in a stack. As the contents of the file was being read, the MD6 sub-hash was stored in the stack and the MD6 compression routine is recursively invoked for the next level when the size of the buffer for the current level equals the designated MD6 buffer size (i.e., the MD6 hash for the root of particular quartenary sub-tree is calculated when the MD6 hash has been calculated for all its child nodes). The sub-hash only contains 16 MD6 words. When the number of children for a particular Merkle sub-tree is less than 4, the rest of the buffer is padded with zeroes to make up for the designated buffer size. The hash storing mechanism of the recursive version of MD6 shown in Figure 5.1.

5.2.1.1 Pseudocode for the Recursive version of MD6

The pseudocode of the recursive MD6 is described as follows:

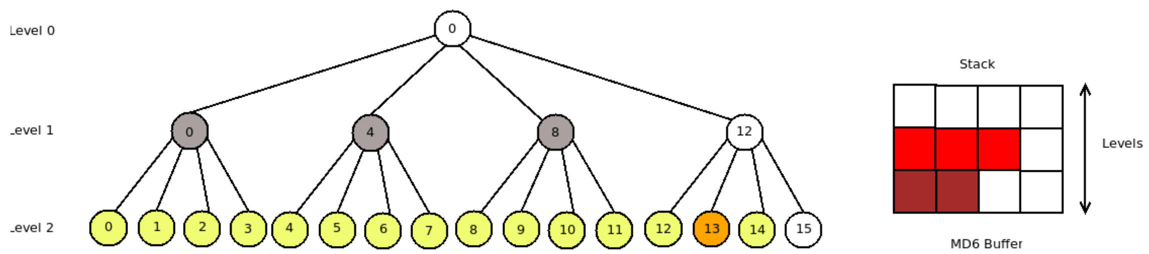

Figure 5.1: Hash storing mechanism in Recursive MD6: In this example, MD6 sub-hash computation for the 13th node is in progression. The status of the stack is shown. Each highlighted tile in the stack represents a sub-hash of size - 16 MD6 words. At this point, 3 nodes at level 1 have their sub-hash inputs. The final hash will be computed after the 4th node at level 1 receives it sub-hash input.

Input:

data[BUFFERS] - Stores the data in chunks of a predetermined size.

Stack[Levels] - A stack to store the sub-hashes for each level.

Childs - 4 (Quartenary Tree)

Output:

The final hash value is stored in the last index of Stack.

Procedure: RECURSIVE MD6

START: Perform on the current buffer

i. Invoke *MD6 compression function* on current buffer

ii. Append sub-hash in the next level of Stack

iii. Set level to the current level of the tree

iv. if number of children equals Childs do

Invoke **RECURSIVE MD6** on Stack[level]

Append sub-hash in the next level of Stack

Reset on Stack[level]

Set level = level+1

else return

end if

BUFFERS = (BUFFERS/Childs)

STOP

if level not equal to Levels-1 /* Final merkle sub-tree has less than 4 child nodes */

Invoke *MD6 compression function* on Stack[level]

Append the calculated sub-hash to Stack[Levels-1]

end if

5.2.2 Iterative MD6

As a first step towards parallelizing MD6 for the GPGPUs, the whole program was rewritten as an iterative algorithm. In this new version, the data from the source file was read into a data buffer. The MD6 compression function was then invoked on each of these data chunks, one level at a time. The primary objective of this operation was to make the algorithm suited for parallelism.

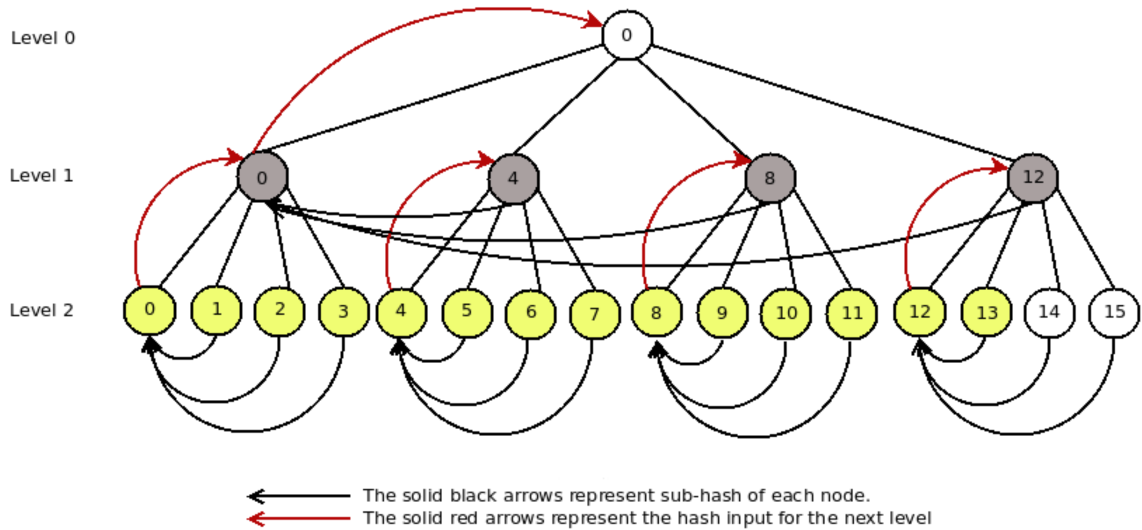

Figure 5.2: Hash storing mechanism in Iterative MD6

The compression routine performed on each chunk of data is independent. This makes it suitable for parallelizing the algorithm for the CUDA architecture.

The MD6 Merkle tree are built on quartenary trees. "Childs" parameter in the algorithm essentially signifies the maximum number of children in a Merkle sub-tree. When the number of children for a particular Merkle sub-tree is less than 4, the rest of the buffer is padded with zeroes to make up for the designated buffer size. The final hash value is stored in the very first index of data. The hash storing mechanism of the implementation of the iterative version of MD6 is shown in Figure 5.2

5.2.2.1 Pseudocode for the Iterative version of MD6

The pseudocode of the iterative MD6 is described as follows:

Input:

data[BUFFERS] - Stores the data in chunks of a predetermined size.

Childs - 4 (Quartenary Tree)

Output:

The final hash value is stored in the first index of data.

Procedure: ITERATIVE MD6

loop until level is final do

Set index = offset to Childs^{level}

Exit loop if it is the final level

for index from 0 to BUFFERS-1 do

i. Invoke *MD6 compression function* on data[index]

ii. Reset data[index % Childs]

iii. Append the calculated sub-hash to data[index % Childs]

end for

Increment level

end loop

5.3 CUDA implementation of MD6

Large inputs allows for more potential parallelism within the compression function computation. In MD6, each level of the tree and all 16 steps within a single round can be computed at once in parallel. Therefore, this function was chosen as the kernel function for the CUDA implementation. Each CUDA block essentially invoked the MD6 compression function on one data block with 16 threads performing the 16 steps of the compression round. The number of CUDA blocks was limited to the size of the source file and finally to the physical memory of the GPU device.

5.3.1 Description of Host Function

The host function essentially reads the data from the source file into the MD6 buffers, initializes the state variables, allocates the GPU memory for performing the CUDA operations on the device side and invokes the MD6 compression function on each of the data buffers. A snapshot of the CUDA MD6 host function is given in Figure A.1. It can be noted that the grid size is designated to be the total number of data blocks of the predetermined CUDA MD6 buffer size(64 words).

5.3.2 Description of Kernel Function

The kernel function eventually runs the MD6 compression routine on each data buffer. The index of the data block directly corresponds to the CUDA block id and each CUDA block executes 16 threads which directly corresponds to the 16 steps performed in each compression round. Each thread accesses an independent data element in the buffer and hence the kernel is embarrassingly parallel in nature. Figure A.2 shows the kernel function execution one round of compression on 16 different CUDA threads. The index for each element in the buffer is calculated by adding the x- and y- indices of the thread block. There are three kernel functions in the CUDA implementation of MD6. From Figure A.1, the following functions should be noted:

- Function **md6_compress_block()** does the preprocessing using the auxillary inputs as discussed in Section 4.2.3.
  <<< Grid size, Thread blocks>>> = <<< Number of data blocks, 1>>>

- Function **md6_compress()** invokes the md6 compression routine on the device, refer Section 5.3.3.
  <<< Grid size, Thread blocks>>> = <<< Number of data blocks, 16>>>

- Function **md6_rewrite()** stores the sub-hashes of each thread block at the appropriate indices.
  <<< Grid size, Thread blocks>>> = <<< (Number of data blocks/4), 1>>>

Each of these functions execute in parallel with **size of grid = total number of MD6 buffers**. It should be noted that function **md6_rewrite()** essentially performs the same task as shown in step (iii) of the non-parallel iterative MD6 version.

5.3.3 Pseudocode for the CUDA implementation of MD6

The pseudocode for the CUDA implementation of MD6 is given as follows:

Input:

N : An input array $N[0..n-1]$ of n = 89 words.

r : A non-negative number of rounds.

Output:

C: An output array $C[0..c-1]$ of length c = 16 words.

State variables:

Number of blocks - Depends on the size of the input file.

Number of threads - 16.

MD6 word size - 8 Bytes

MD6 buffer size - Variable (default size - 512 Bytes)

Procedure: CUDA md6 kernel

For each block do

Set $index$ to $blockID$

For each $data[index]$ do

Set i to $n + ThreadID$: /* 16 steps */

$x = S_{i-n} \oplus A_{i-n} \oplus A_{i-t_0}$

$x = x \oplus (A_{i-t_1} \wedge A_{i-t_2}) \oplus (A_{i-t_3} \wedge A_{i-t_4})$

$x = x \oplus (x \gg r_{i-n})$

$x = x \oplus (x \ll r_{i-n})$

exit CUDA block

exit CUDA kernel call

5.4 CUDA MD6 for file matching

CUDA MD6 can perform file matching in two modes:

- Direct Hashing
- Recursive Hashing

Recursive hashing generates file hashes by recursively working through a folder of files. The program can also find absolute file-matches provided with a predetermined list of hashes. The hash matching algorithm is a normal string matching procedure. If the hash was not found in the input hash list, the new hash will be appended to the list. In case of working with larger files, CUDA MD6 effectively partitions the content of the large files into adequate data chunks, calculating the MD6 hashes of each chunk and invoking the MD6 algorithm on those hashes for the final time to

compute the output. The physical size of the data chunk is limited by the physical memory of the GPU device.

Chapter 6

EVALUATION OF CUDA MD6

This chapter discusses the various experiments done using CUDA MD6. The final section provides a general discussion on improving the GPU performance.

6.1 Experiment 1: Executing CUDA MD6 on single files

To test the speedup of the parallel MD6 over the serial version, the program was executed on a range of files in increasing order of sizes. The increase in speedup followed a geometric trend with the increase in size of the files. The results are shown in Figure 6.1.

6.1.1 Analysis of Experiment 1

It can be noted from the Figure 6.1, that the speedup is more pronounced in the case of CUDA MD6 execution on larger files. The speedup graph exhibits a geometric trend. It should also be noted that the startup time for the device (Initialization) is almost constant when CUDA MD6 is run on files of any size. This explains the relative lower speedup when CUDA MD6 is run on smaller files. It is evident from Figure 6.1 that significant speedup is achieved when the file input size exceeds a certain threshold. With a default CUDA buffer size of 512 bytes and the number of compression rounds being 72, this threshold was found to be approximately 20MB.

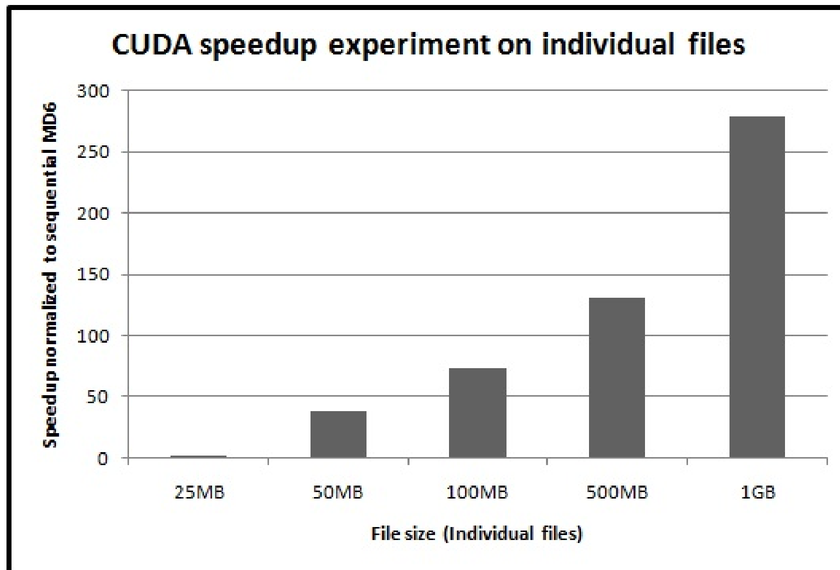

Figure 6.1: CUDA speedup experiment on individual files (Direct Hashing): The program was executed on individual files of sizes ranging from 25 MB to 1GB with the MD6 Buffer size preset to 512 bytes and the number of compression rounds = 72.

6.2 Experiment 2: Executing CUDA MD6 on archive of files

For the second experiment, the program was executed on the archive of collective file sizes of the range from 100MB to 8GB. The results are depicted in diagram Figure 6.2.

6.2.1 Analysis of Experiment 2

It can be noted from Figure 6.2, that the speedup is relatively low as compared to Experiment 1. However, this experiment is highly subjective to the number of files in the archive and their physical size. For instance, a folder with small number of files and a larger total physical size would give a relatively higher speedup compared to that of a folder with large number of files and smaller physical size.

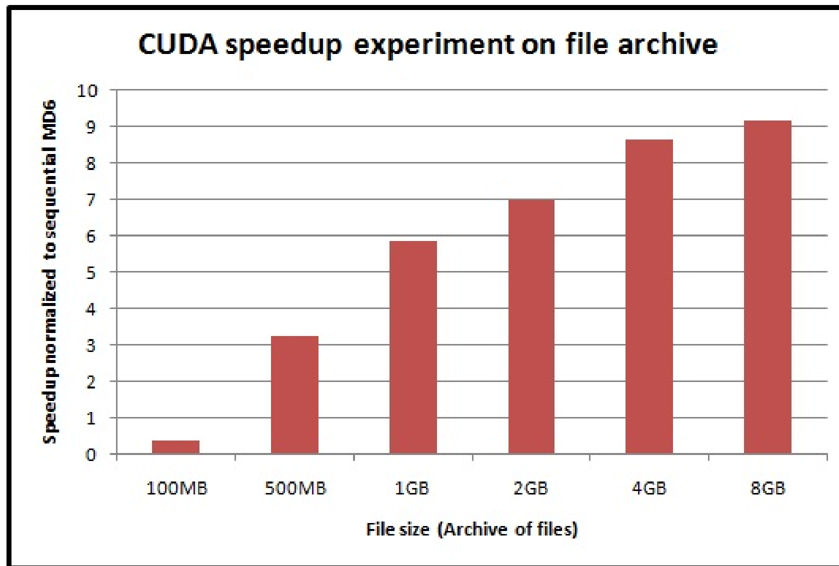

Figure 6.2: CUDA speedup experiment on archive of files (Recursive Hashing): The program was executed individually each time on an archive of files of sizes ranging from 100 MB to 8GB with the MD6 Buffer size preset to 512 bytes.

6.3 Experiment 3: Executing CUDA MD6 with varying MD6 Buffer size

The parallel version of MD6 was executed on these files with a default buffer size of 512 Bytes. As we presumed in the previous section, the speedup was found to scale up with the size of the file. To generate the speedup graphs, the execution time of serial and parallel MD6 was averaged over 10 runs. The speedup graphs were also generated for different buffer sizes. The speedup follows a linear trend with the increase in the size of the buffers. For example, the speedup of the parallel **MD6** executed with buffer sizes of 512 Bytes, 1KB, 1.5KB and 2KB on a 100MB file is shown in Figure 6.3.

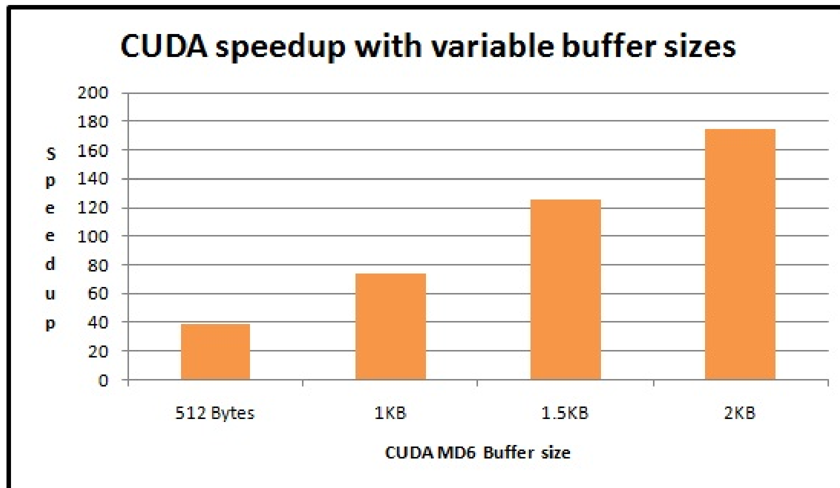

Figure 6.3: Real time speedup of CUDA MD6 with varying MD6 buffer sizes: CUDA MD6 was executed with varying CUDA buffer sizes of 512 Bytes, 1KB, 1.5KB and 2KB on a 100MB source file.

6.3.1 Analysis of Experiment 3

From the figure, it follows that the size of the Merkle-tree shrinks by a factor of N, when the MD6 buffer size is increased by N. This explains the increasing linear trend in speedup of a factor of N(approximated).

6.4 General Analysis

The speedup with and without the CUDA overhead is tabulated in Table 6.1. The speedup comparisons done for experiment 1 with three different filesizes 100MB, 500MB, 1GB is shown graphically in Figure 6.4.

It can be observed from Table 6.1 that the average real time speedup is severely impeded by the CUDA overhead. The actual execution time of the device is only 5% of the real execution time. The real time execution speedup of the parallel MD6 over the serial version is lower than the actual execution time of the CUDA threads due to the overhead incurred by various CUDA operations like global memory allocation, host-to-device copy of data and finally device-to-host copy overhead. With the increase in the number of compression rounds, the CUDA MD6 speedup increases

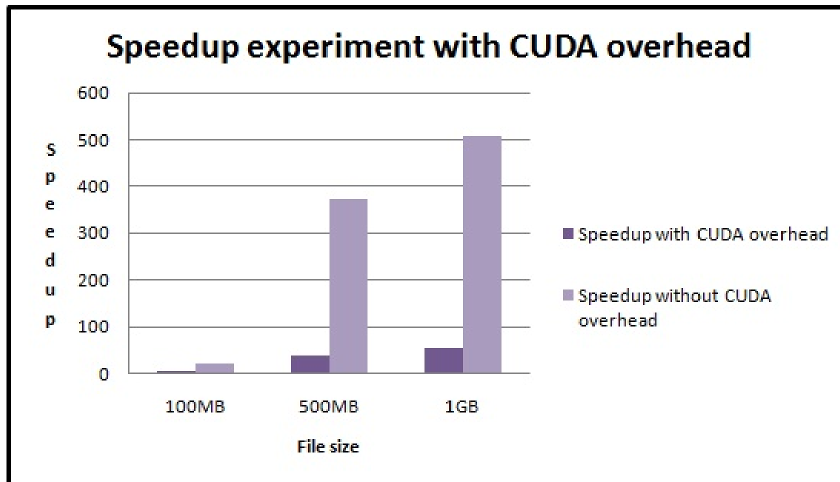

Figure 6.4: CUDA Speedup comparison with and without CUDA overhead (Graph): The graph shows the speedup comparison with and without CUDA overhead for the direct hashing experiment.

File/Archive size	Speedup with CUDA overhead	Speedup without CUDA overhead
100MB	2.75	19.87
500MB	38.89	373.98
1GB	52.72	506.78

Table 6.1: CUDA Speedup comparison with and without CUDA overhead for the direct hashing experiment

correspondingly. This is given in Figure 6.5. It can be observed that the file size threshold is around 100MB for a single round of compression. It has to be noted that the integrity of the checksum depends on the number of compression rounds. The integrity of the checksum improves with the increase in the number of compression rounds.

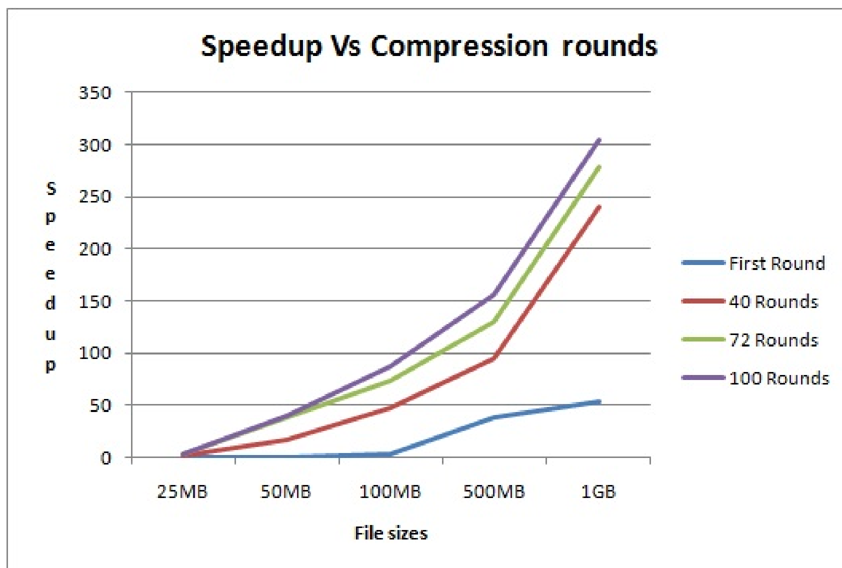

Figure 6.5: CUDA Speedup with different number of compression sounds: The graph shows the speedup comparison with varying number of compression rounds on different file sizes.

Chapter 7

CONCLUSION AND FUTURE WORK

This chapter presents the conclusion and the description of the future work that remains to be done.

7.1 Conclusion

In this thesis we implemented a file matching application for identifying similar files. The hashing algorithm used for the design of this application is MD6 which belongs to the line of message digest algorithms. An important feature of the MD6 algorithm is that it is very well suited for portability on the GPU's. The md6 compression function which performs the brunt of computation is highly parallelizable and it was chosen as the kernel function. The kernel function is the parallelizable function that executes on the GPU device. The algorithm was first rewritten as a parallelizable iterative version from the serial stack-based recursive version. The iterative version of MD6 was then mapped to the GPU, by intelligently designing the partition procedure and also exploiting the multi-core architecture of the GPU's. The application was extended to incorporate the functionality of absolute file matching thus making it a full-fledged file matching application. CUDA MD6 can eventually perform recursive hashing on archive of files and identify similar files. Due to the rapid execution of CUDA MD6, it can essentially be used as a file matcher, integrity checker or a checksum calculator. Hence, we conclude that CUDA MD6 is one of the solutions for faster file matching on a large scale.

7.2 Future work

We can identify the following tasks that can be considered for future work.

- The internal data structures of the **CUDA MD6** kernel can be modified to exploit the texture memory. The texture memory access in the CUDA architecture is many-fold faster than the global memory access.
- The use of page-locked memory can be utilized in the kernel for exploiting the higher bandwidth. It may facilitate faster streaming during the data transfer operations between the host and the device.
- RAM Disks can be utilized for faster data transfer from the host to the device.
- The effectiveness of other GPU implementations like **CUDA MD5** over **CUDA MD6** can be analyzed.
- **CUDA MD6** can be incorporated to **HASHDEEP** which provides a suite of different algorithms for file matching.
- A graphical user interface can be designed to allow end users to operate **CUDA MD6** in a easier fashion.
- **CUDA MD6** is written for the linux environment. It can be ported to other platforms like Windows, Mac OS.

BIBLIOGRAPHY

[1] Behrooz Parhami. "*Introduction to Parallel Processing (Algorithms and Architectures).*" Kluwer academic publishers, 1999.

[2] The official resource for CUDA programming and applications. http://www.nvidia.com/object/cuda_home_new.html

[3] Abraham Silberschatz, Greg Gagne and Peter Baer Galvin. "*Operating System Concepts*", 8th Edition, 2009.

[4] David A.Patterson and John L.Hennessy. "*Computer Organization and Design*", Elsevier Inc, 2009.

[5] ATI Stream Developer showcase. "http://developer.amd.com/samples/streamshowcase/Pages/default.aspx"

[6] Ronald Rivest. "*The MD5 Message-Digest Algorithm*". RFC 1321, 1992.

[7] Donald Eastlake and Paul Jones (Cisco Systems). "*US Secure Hash Algorithm 1 (SHA1)*", RFC 3174, September 2001.

[8] National Institute of Standards and Technology."*Secure Hash Standard*", Federal Information Processing Standards Publication, Volume 180, Issue 2, August 1, 2002.

[9] Ross Anderson and Eli Biham. "*Tiger: A Fast New Hash Function*".

[10] Vincent Rijmen and Paulo Barreto. "*The WHIRLPOOL hash function*". World-Wide Web document, 2001.

[11] Ronald Rivest, Adi Shamir and Leonard Adleman. "*A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*", Communications of the ACM 21.

[12] Jesse Kornblum. "*HASHDEEP File matching program*". http://md5deep.sourceforge.net/hashdeep.html

[13] Jesse Kornblum. "*SSDEEP File matching program*". http://ssdeep.sourceforge.net/

[14] Jesse Kornblum. "*Identifying almost identical files using context triggered piecewise hashing*",Digital Investigation, Volume 3, Supplement 1, Pages 91-97, September 2006.

[15] NVIDIA Inc. "*NVIDIA CUDA Programming Guide (Version 3.0)*", Appendix B,Section B.1.4,Page 108.

[16] Ivan Bjerre Damgard. "*A design principle for hash functions.*", Advances in Cryptology CRYPTO 89, pages 416-427, Springer, August 1990.

[17] Alfred J.Menezes, Paul C.van Oorschot and Scott A.Vanstone. "*Handbook of Applied Cryptography.*" CRC Press, 1997. http://www.cacr.math.uwaterloo.ca/hac/.

[18] Ralph C.Merkle. "*One way hash functions and DES.*" In Advances in Cryptology CRYPTO 89, pages 428-446. Springer,1990.

[19] Ronald L.Rivest. "*The MD6 hash function.*" Advances in Cryptology - CRYPTO 08.

[20] Christopher Yale Crutchfield. "*Security proofs for the MD6 hash function mode of operation.*" Masters thesis, MIT EECS Department, 2008. http://groups.csail.mit.edu/cis/theses/crutchfield-masters-thesis.pdf.

[21] Information on CUDA enabled GPUs. http://www.nvidia.com/object/cuda_learn_products.html.

[22] CUDA Software and NVIDIA driver download site. http://www.nvidia.com/object/cuda_get.html.

[23] Andrew Tridgell. Spamsum README. http://samba.org/ftp/unpacked/junkcode/spamsum/README, 2002.

[24] Andrew Tridgell. "*Efficient algorithms for sorting and synchronization.*" PhD thesis. Canberra, Australia: Department of Computer Science, The Australian National University, 1999.

[25] Klayton Monroe and Dave Bailey. "*FTimes - a system baselining and evidence collection tool.*" http://ftimes.sourceforge.net/FTimes/index.shtml

[26] Changxin Li, Hongwei Wu, Shifeng Chen, Xiaochao Li and Donghui Guo. "*Efficient implementation for MD5-RC4 encryption using GPU with CUDA.*" In the proceedings of Anti-counterfeiting, Security, and Identification in Communication, August 2009. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5247340

[27] Janaka Deepakumara, Howard M. Heys, and Venkatesan. "*FPGA Implementation of MD5 Hash Algorithm.*" In the Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE 2001), Toronto, Canada, May 2001.

[28] Kimmo Jarvinen, Matti Tommiska and Jorma Skytta. "*Hardware Implementation Analysis of the MD5 Hash Algorithm.*" In the Proceedings of the 38th Annual Hawaii International Conference on System Sciences, September 2005.

[29] Diez, Bojani, Stanimirovic, Carreras and Nieto-Taladriz. "*Hash Algorithms for Cryptographic Protocols: FPGA Implementations.*" In the Proceedings of the 10th Telecommunications Forum, TELFOR2002, Belgrade, Yugoslavia, November 2002.

[30] Sandra Dominikus. "*A Hardware Implementation of MD4-Family Hash Algorithms.*" Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2002), Dubrovnik, Croatia, Volume 3, Pages:1143 1146, September 2002.

[31] Kurt K.Ting, Steve C.L.Yuen, Lee and Philip Heng Wai Leong. "*An FPGA Based SHA-256 Processor.*" Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream, Springer Volume 2438/2002, Pages:449-471, 2002.

[32] Anton Bosselaers, Rene Govaerts and Joos Vandewalle. "*Fast Hashing on Pentium.*" Advances in Cryptology - CRYPTO'96.

[33] Imtiaz Ahmad and Shoba Das. "*Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs.*" Volume 31, Issue 6, Pages 345-360, Elsevier, September 2005,

[34] Junko nakajima and Mitsuru matsui. "*Performance analysis and parallel implementation of dedicated hash functions.*" In the Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques:Advances in cryptology, Pages: 165-180, 2002.

[35] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. "*Improving sha-2 hardware implementations.*" In Workshop on Cryptographic Hardware and Embedded Systems (CHES 2006).

[36] Maire McLoone and Ciaran McIvor. "*High-speed low area hardware architectures of the whirlpool hash function.*" Journal of VLSI Signal Processing Systems, Volume 47, Issue 1, Pages:4757, May 2007.

[37] Haralambos E.Michail, Athanasios P.Kakarountas, George Theodoridis and Costas E.Goutis. "*A low-power and high-throughput implementation of the sha-1 hash function.*" In ICCOMP05: Proceedings of the 9th World Scientific and Engineering Academy and Society (WSEAS) International Conference on Computers, Stevens Point, Wisconsin, USA, Pages: 1-6, 2005.

[38] Nicolas Sklavos and Odysseas G.Koufopavlou. "*Implementation of the sha-2 hash family standard using fpgas.*" The Journal of Supercomputing, Volume 31, Issue 3, Pages:227-248, March 2005.

[39] Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. "*A compact fpga implementation of the hash function whirlpool.*" In the Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays, pages 159-166, New York, NY, USA, May 2006.

[40] Kitsos and Odysseas G.Koufopavlou. "*Efficient architecture and hardware implementation of the whirlpool hash function.*" IEEE Transactions on Consumer Electronics, Volume 50, Issue 1, Pages:218-213, Feb 2004.

[41] Johnny Bengtsson. "*Parallel Password Cracker: A Feasibility Study of Using Linux Clustering Technique in Computer Forensics*", Digital Forensics and Incident Analysis, 2007.

[42] Ryan Lim. "*Parallelization of John the Ripper (JtR) using MPI*" Apr 9, 2009. http://www.ryanlim.com/personal/jtr-mpi/report.pdf

[43] Andrew Zonenberg. "*Distributed Hash Cracker: A Cross-Platform GPU-Accelerated Password Recovery System.*" Rensselaer Polytechnic Institute, 2009.

[44] Ivan Golubev. "*IGHASHGPU*". http://www.golubev.com/hashgpu.htm.

[45] Lin Zhou and Wenbao Han, "*A Brief Implementation Analysis of SHA-1 on FPGAs, GPUs and Cell Processors*", 2009 International Conference on Engineering Computation (icec), Pages:101-104, 2009.

[46] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan and Matei Ripeanu. "*StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems*", In the proceedings of ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC 2008), Boston, MA, June 2008.

[47] Stanley Tzeng and Li-Yi Wei. "*Parallel white noise generation on a GPU via cryptographic hash.*" Symposium on Interactive 3D Graphics archive, In the Proceedings of the 2008 symposium on Interactive 3D graphics and games, Pages: 79-87, 2008

[48] Open source forensic tools. http://www.opensourceforensics.org/tools/unix.html.

Appendix A

APPENDIX

A.1 Host component

A.2 Kernel component

```
#include "md6.h"

typedef struct {
        int d;           /* desired hash bit length. 1 <= d <= 512.      */
        int hashbitlen;  /* hashbitlen is the same as d; for NIST API    */
        unsigned char hashval[ md6_c*(md6_w/8) ];
        unsigned char hexhashval[(md6_c*(md6_w/8))+1];
        md6_word K[ md6_k ];
        int keylen;
        int L;
        int r;
} md6_state;


int devicecount;
int d;                    /* digest length */
int L;                    /* mode parameter */
int r;                    /* number of rounds */
int keylen;               /* key length in bytes (at most 64) */
md6_state st;             /* md6 computation state */


char input_hashes[100][100];
char input_files[100][50];
char output_hashes[100][100];
int in_counter = 0;
int out_counter = 0;
char help_string[] =
"To execute CUDA MD6, run\n"
"MD6 [INPUT HASH FILE] [FOLDER / FILE]"
"[INPUT HASH FILE] is optional. It is the file that contains predetermined hashes"
"CUDA MD6 recursively processes folders or individual files."
;

int BUFFERS;
md6_word *Data = NULL;
extern "C"
double computeMD6(md6_state *,int,int,int,int,md6_word *,int);

/*
 *      Function: hash_init
 *
 *      Description: MD6 state variables initialization
 *
 */

void hash_init(){
        memset(&st,0,sizeof(md6_state));  /* clear state to zero */
        st.d = d;                        /* save hashbitlen */
        st.keylen = 0;
        st.L = L;
        st.r = r;
}

/*
 *      Function: flush
 *
 *      Description: Free the MD6 host buffer
 *
 */

void flush(){
        free(Data);
}

/*
 *      Function: read_file
```

Figure A.1: Host component: md6sum.cpp

```
 *
 *      Description: Read the contents of the source file into the host MD6 buffer
 *
 */

int read_file(char *filename){
        uint64_t bytes;
        int count = 0;
        unsigned long int no_of_runs=1;
        int MD6_words = 64;
        unsigned char temp[512];
        BUFFERS = 5000000;
        FILE *inFile = fopen (filename, "rb");
        if ( inFile == NULL )  {
                printf("\n%s can't be opened.\n", filename);
                return 0;
        }
        Data = (md6_word *)malloc(MD6_words*sizeof(md6_word)*BUFFERS);
        if(!Data) {
                printf("\nNo memory!");
                fclose(inFile);
                exit(0);
        }
        while ((bytes = fread (temp, 1, 512, inFile)) != 0){
                memcpy(&Data[count*MD6_words],temp,512);
                count++;
                if((count/40960)>no_of_runs-1)
                no_of_runs++;
        }
        BUFFERS = count;
        fclose(inFile);
        return no_of_runs;
}

/*
 *      Function: process_hash
 *
 *      Description: Reads the predetermined set of hashes into a temporary buffer
 *
 */

void process_hash(char *filename){
        int count = 0;
        FILE *hash_file = fopen (filename, "r");
        if ( hash_file == NULL )  {
                printf("\n%s can't be opened.\n", filename);
                return;
        }else if ( hash_file != NULL ){
                char line [ 100 ]; /* or other suitable maximum line size */
                while ( fgets ( line, sizeof(line), hash_file ) != NULL ) /* read a line */{
                        memcpy(input_hashes[count],line,sizeof(line));
                        count++;
                }
        }
        fclose (hash_file );
        in_counter=count;
}

/*
 *      Function: write_hashes
 *
 *      Description: Performs hash comparison if input hash file is given. Writes the unique hashes to
the output file
 *
 */

void write_hashes(){
```

Figure A.1: Host component: md6sum.cpp

```
        int i,j;
        FILE *outfile = fopen ("hashes.txt", "a");
        if ( outfile == NULL )  {
                return ;
        }else if ( outfile != NULL ){
                if(in_counter!=0){
                        for(i=0;i<in_counter;i++){
                                for(j=0;j<out_counter;j++){
                                        if(strcmp(input_hashes[i],output_hashes[j])==0){
                                                printf("\n Hash match found for file - %s",input_files
[j]);
                                        }else{
                                                fputs ( output_hashes[i], outfile ); /* write the line */
                                                fputs("\n",outfile);
                                        }
                                }
                        }
                }else{
                        for(i=0;i<out_counter;i++){
                                fputs ( output_hashes[i], outfile ); /* write the line */
                                fputs("\n",outfile);
                        }
                }
        }
        fclose (outfile );
}



/*
 *      Function: md6compute_hex_hashval
 *
 *      Description: Perform Hex translation on the final MD6 hash
 *
 */


int md6compute_hex_hashval( md6_state *st ){
        int i;
        static unsigned char hex_digits[] = "0123456789abcdef";
        if ( st == NULL ) return MD6_NULLSTATE;
        for (i=0;i<((st->d+7)/8);i++) {
                st->hexhashval[2*i] = hex_digits[ ((st->hashval[i])>>4) & 0xf ];
                st->hexhashval[2*i+1] = hex_digits[ (st->hashval[i]) & 0xf ];
        }
        st->hexhashval[(st->d+3)/4] = 0;
        return MD6_SUCCESS;
}

/*
 *      Function: trimhashval
 *
 *      Description: Pad partial hashes
 *
 */


void trimhashval(md6_state *st){
        int full_or_partial_bytes = (st->d+7)/8;
        int bits = st->d % 8;                   /* bits in partial byte */
        int i;
        for ( i=0; i<full_or_partial_bytes; i++ )
        st->hashval[i] = st->hashval[md6_c*(md6_w/8)-full_or_partial_bytes+i];
        for ( i=full_or_partial_bytes; i<md6_c*(md6_w/8); i++ )
        st->hashval[i] = 0;
        if (bits>0) {
```

Figure A.1: Host component: md6sum.cpp

```
                for ( i=0; i<full_or_partial_bytes; i++ ) {
                        st->hashval[i] = (st->hashval[i] << (8-bits));
                        if ( (i+1) < md6_c*(md6_w/8) )
                        st->hashval[i] |= (st->hashval[i+1] >> bits);
                }
        }
}

md6_word md6byte_reverse( md6_word x ){
        x = (x << 32) | (x >> 32);
        return x;
}


void md6reverse_little_endian( md6_word *x, int count ){
        int i;
        for (i=0;i<count;i++)
        x[i] = md6byte_reverse(x[i]);
}

/*
 *      Function: md6_final
 *
 *      Description: Perform final level translation on the MD6 hash
 *
 */


int md6_final( md6_state *st , unsigned char *hashval){
        md6reverse_little_endian( (md6_word*)st->hashval, md6_c );
        trimhashval( st );
        if (hashval != NULL) memcpy( hashval, st->hashval, (st->d+7)/8 );
        md6compute_hex_hashval( st );
        return MD6_SUCCESS;
}

/*
 *      Function: md6_hash
 *
 *      Description: Perform MD6 compression on the given source file
 *
 */

double hash(char *Dir,char *filename){
        int level,final,pad_bits,no_of_runs;
        double time=0;
        level=final=pad_bits=0;
        char File[50];
        strcpy(File,Dir);
        strcat(File,filename);
        printf("\n Reading file %s",File);
        if(strcmp(filename,".")==0) return 0;
        else if(strcmp(filename,"..")==0) return 0;
        hash_init();
        no_of_runs=read_file(File);
        time = computeMD6(&st,level,final,pad_bits,BUFFERS,Data,no_of_runs);
        memcpy( st.hashval, Data, md6_c*8 );
        md6_final(&st,NULL);
        flush();
        return time;
}
void read_archive(char *file){
        DIR *dp;
        struct dirent *ep;
        double time = 0;
        double total_time = 0;
        int counter =0;
```

Figure A.1: Host component: md6sum.cpp

```
        dp = opendir (file);
        if (dp != NULL){
                while (ep = readdir (dp)){
                        time = hash(file,ep->d_name);
                        if (time==0){ continue;}
                        memcpy(input_files[counter],ep->d_name,sizeof(ep->d_name));
                        total_time += time;
                        printf("\n\nFinal hash value = %s",st.hexhashval);
                        memcpy(output_hashes[counter],st.hexhashval,sizeof(st.hexhashval));
                        counter++;
                }
                (void) closedir (dp);
                printf("\n Total processing time = %f", total_time);

        }else{
                //perror ("Couldn't open the directory");
                total_time = hash(file,"");
                printf("\n\nFinal hash value = %s",st.hexhashval);
                memcpy(input_files[counter],file,sizeof(file));
                memcpy(output_hashes[counter],st.hexhashval,sizeof(st.hexhashval));
                counter++;
                printf("\n Total processing time = %f", total_time);

        }
        out_counter=counter;
        write_hashes();
}


int main(int argc, char **argv){
        int i;
        d = 128;
        keylen = 0;
        L = 128;
        r = 72;
        for (i=1;i<argc;i++)    {
                /* If input hash file is available */
                if (argc==3) {
                        process_hash(argv[i]);
                        i++;
                        read_archive(argv[i]);
                }
                if (argv[i][0]!='-'){
                        read_archive(argv[i]);
                }else {
                        switch ( argv[i][1] ){
                                case 'h': printf("%s",help_string); break;
                                default : printf("%s",help_string); break;
                        }
                }

        }
        return 0;
}
```

Figure A.1: Host component: md6sum.cpp

```
#include "md6.h"
#include <md6_kernel.cu>

/*
        Function: no_of_levels

        Description: Function to calculate the number of levels in the Merkle tree
*/

int no_of_levels(unsigned long int buffers){
        int i,j;
        i=4;
        j=1;
        while(1){
                if(i >= buffers){
                        printf("\n Buffers = %d",buffers);
                        printf("\n Number of levels = %d",j);
                        break;
                }
                i = i*4;
                j++;
        }
        return j;
}

/*
        Function: computeMD6

        Description: A helper to export the kernel call to C++ code
*/

extern "C"
double computeMD6(md6_state* st,int level,int final,int pad_bits,unsigned long int BUFFERS,md6_word
*data,int partitions){
        uint64_t temp = (uint64_t)0x0123456789abcdefULL;
        md6_word *d_idata;
        md6_word *Data[1000];
        md6_word *N;
        uint64_t *S1;
        int i,j,levels;
        int Grid;
        double leaves;
        unsigned int num_blocks;
        unsigned int timer = 0;
        double time = 0;
        int no_of_words = 64;
        int tot_words = 105;
        int MD6_BUFFER_SIZE = 512;
        /* CUDA memory allocation */
        cudaSetDevice( cutGetMaxGflopsDeviceId() );
        cutilSafeCall( cudaMalloc( (void**) &d_idata,BUFFERS*no_of_words*sizeof(md6_word)));
        cutilSafeCall( cudaMalloc( (void**) &N,BUFFERS*tot_words*sizeof(md6_word)));
        cutilSafeCall( cudaMalloc( (void**) &S1, sizeof(uint64_t)));
        cutilSafeCall( cudaMemcpy( S1, &temp, sizeof(uint64_t),cudaMemcpyHostToDevice) );

        /*
                 Invoke kernel for "partitions" number of times
                 partitions ==> Signifies the number of partitions of the Merkle tree when the
                                number of CUDA MD6 buffers >= MAX
                        MAX ==> Signifies the maximum number of CUDA MD6 buffers (or) CUDA Thread blocks
                                to ensure thread safety

        */

        printf("\n TOTAL NUMBER OF PARTITIONS = %d, BUFFERS = %d\n",partitions,BUFFERS);
        for(j=0;j<=partitions;j++){
```

Figure A.1: Host component: execute_kernel.cu

```
		printf("\n -------------- EXECUTING PARTITION %d -----------------",j+1);
		/* Copy all the sub-hashes for a final run if the number of buffers is greater than or 
equal to MAX */
		if((j==partitions)&&(BUFFERS>=MAX)){
			printf("\n Copying the sub-hashes of %d partitions for the final run",partitions);
			/* Copy all the sub-hashes for each Merkle sub-tree of size <= MAX */
			for(i=0;i<partitions;i++)
			cutilSafeCall( cudaMemcpy(d_idata+(i*MD6_BUFFER_SIZE), Data[i],no_of_words*sizeof
(md6_word),cudaMemcpyHostToDevice) );					BUFFERS=partitions;
			levels = no_of_levels(partitions);
		}else if((j==partitions)&&(BUFFERS<MAX)){

			/* If number of buffers is lesser than MAX, then the final hash as already been 
calculated */
			break;
		}else{
			if(BUFFERS>=MAX){
				/* If number of buffers is greater than or equal to MAX */
				if(j==partitions-1){	/* Copy the last sub-tree */
					levels = no_of_levels((BUFFERS-(j*MAX)));
					cutilSafeCall(cudaMemcpy(d_idata, data+(j*MAX*MD6_BUFFER_SIZE),
(BUFFERS-(j*MAX))*no_of_words*sizeof(md6_word),cudaMemcpyHostToDevice) );

				}else{  /* Copy the sub-tree of maximum size */
					cutilSafeCall(cudaMemcpy(d_idata, data
+(j*MAX*MD6_BUFFER_SIZE),MAX*no_of_words*sizeof(md6_word),cudaMemcpyHostToDevice) );
					levels = no_of_levels(MAX);
				}
			}else{
				/* If number of buffers is lesser than MAX */
				cutilSafeCall(cudaMemcpy(d_idata, data,BUFFERS*no_of_words*sizeof
(md6_word),cudaMemcpyHostToDevice) );
				levels = no_of_levels(BUFFERS);
			}
		}


		/* CUDA kernel invocation for a Merkle sub-tree of size <= MAX */
		for(i=0;i<levels;i++){

			/* Start CUDA timer */
			cutilCheckError(cutCreateTimer(&timer));
			cutilCheckError(cutStartTimer(timer));
			leaves = int(pow(4,i+1));

			/* Invoke md6_pack routine on each MD6 buffer */
			if (BUFFERS >= MAX){
				if(j==partitions -1){
					num_blocks = int(ceil((BUFFERS-(j*MAX))/pow(4,i)));
					Grid = int(ceil((BUFFERS-(j*MAX))/double(leaves)));

				}else {
					num_blocks = int(ceil(MAX/pow(4,i)));
					Grid = int(ceil(MAX/double(leaves)));
				}
			}else{
				num_blocks = int(ceil(BUFFERS/pow(4,i)));
				Grid = int(ceil(BUFFERS/double(leaves)));
			}

			/* Invoke md6_pack routine on each MD6 buffer */
			printf("\n\nKernel call 1: Number of grids = %d",num_blocks);
			dim3  grid( num_blocks, 1, 1);
			dim3  threads( 1, 1, 1);
			md6_compress_block<<<grid, threads>>>(S1,d_idata,(leaves/4),N);
			cudaThreadSynchronize();
```

Figure A.1: Host component: execute_kernel.cu

```
                        /* Invoke md6 compression routine on each MD6 buffer */
                        printf("\nKernel call 2: Number of grids = %d",num_blocks);
                        dim3 grid1(num_blocks,1,1);
                        dim3  threads1( 16, 1, 1);
                        md6_compress<<<grid1,threads1>>>(S1,d_idata,N,(leaves/4));
                        cudaThreadSynchronize();

                        /* Store the calculated sub-hash values at the appropriate indices for the next
level */
                        printf("\nKernel call 3: Number of grids = %d",Grid);
                        dim3  grid2(Grid, 1, 1);
                        dim3  threads2( 1, 1, 1);
                        md6_rewrite<<<grid2,threads2>>>(d_idata,(leaves/4));
                        cudaThreadSynchronize();

                        /* Stop CUDA timer */
                        cutilCheckError(cutStopTimer(timer));
                        time += cutGetTimerValue(timer);
                        printf("\nProcessing time: %f (ms) \n", time);
                        cutilCheckError(cutDeleteTimer(timer));
                }

                /* Store all the sub-hashes in a temporary buffer */
                if(j!=partitions){
                        Data[j]=(md6_word *)malloc(no_of_words*sizeof(md6_word));
                        printf("\n Writing to the %d th buffer",j);
                        cutilSafeCall( cudaMemcpy(Data[j], d_idata,no_of_words*sizeof
(md6_word),cudaMemcpyDeviceToHost));
                }
        }

        /* CUDA cleanup calls */
        printf(cudaGetErrorString(cudaGetLastError()));
        cutilSafeCall( cudaMemcpy(data, d_idata,no_of_words,cudaMemcpyDeviceToHost));
        cutilSafeCall(cudaFree(d_idata));
        cutilSafeCall(cudaFree(S1));
        cutilSafeCall(cudaFree(N));
        cudaThreadExit();

        /* Free all the memory in the temporary buffer */
        for(i=0;i<partitions;i++){
                free(Data[i]);
        }
        return time;
}
```

Figure A.1: Host component: execute_kernel.cu

```
#include "md6.h"
#ifndef _TEMPLATE_KERNEL_H_
#define _TEMPLATE_KERNEL_H_

typedef uint64_t md6_word;
typedef uint64_t md6_control_word;
typedef uint64_t md6_nodeID;
typedef struct {
        int d;           /* desired hash bit length. 1 <= d <= 512.      */
        int hashbitlen;  /* hashbitlen is the same as d; for NIST API    */
        unsigned char hashval[ md6_c*(md6_w/8) ];
        unsigned char hexhashval[(md6_c*(md6_w/8))+1];
        md6_word K[ md6_k ];
        int keylen;
        int L;
        int r;
} md6_state;

__device__
static const md6_word Q[15] =
{
    0x7311c2812425cfa0,
    0x6432286434aac8e7,
    0xb60450e9ef68b7c1,
    0xe8fb23908d9f06f1,
    0xdd2e76cba691e5bf,
    0x0cd0d63b2c30bc41,
    0x1f8ccf6823058f8a,
    0x54e5ed5b88e3775d,
    0x4ad12aae0a6d6031,
    0x3e7f16bb88222e0d,
    0x8af8671d3fb50c2c,
    0x995ad1178bd25c31,
    0xc878c1dd04c4b633,
    0x3b72066c7a1552ac,
    0x0d6f3522631effcb,
};


#define  t0   17     /* index for linear feedback */
#define  t1   18     /* index for first input to first and */
#define  t2   21     /* index for second input to first and */
#define  t3   31     /* index for first input to second and */
#define  t4   67     /* index for second input to second and */
#define  t5   89     /* last tap */

#define w 64     /* # bits in a word                  (64) */
#define n 89     /* # words in compression input      (89) */
#define c 16     /* # words in compression output     (16) */
/* MD6 constants needed for mode of operation                  */
#define q 15     /* # words in Q                      (15) */
#define k 8    /* # words in key (aka salt)         (8)  */
#define u 1     /* # words in unique node ID         (1)  */
#define v 1     /* # words in control word           (1)  */
#define b 64     /* # data words per compression block (64) */

#ifndef min
#define min(a,b) ((a)<(b)? (a) : (b))
#endif
#ifndef max
#define max(a,b) ((a)>(b)? (a) : (b))
#endif
```

Figure A.2: Kernel component: md6_kernel.cu

```
__device__
int N1 = 89;

__device__
uint64_t S;

/*
        Device Function: loop_body

        Description: MD6 specific computation
*/

__device__
void loop_body(md6_word* A,int rs,int ls,int step,unsigned long long int x,int i)
{

      x ^= A[i+step-t5];                   /* end-around feedback   */

      x ^= A[i+step-t0];                   /* linear feedback       */

      x ^= ( A[i+step-t1] & A[i+step-t2] ); /* first quadratic term  */

      x ^= ( A[i+step-t3] & A[i+step-t4] ); /* second quadratic term */

      x ^= (x >> rs);                      /* right-shift           */

      A[i+step] = x ^ (x << ls);           /* left-shift            */
}

/*
        Device Function: md6_main_compression_loop

        Description: 16 independent CUDA threads compute on the MD6 Buffer corresponding to each block
*/

__device__
void md6_main_compression_loop(md6_word *A,uint64_t *S0,int R,int C)
{
    int step,index;
    unsigned long long int Smask = 0x7311c2812425cfa0ULL;
    int W=64;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    S = *S0;
    step = tx+ty;
    index = 89;
    /* ***************************************************************** */
        switch(step){
                case 0:
                        loop_body(A,10,11,step,S,index);break;
                case 1:
                        loop_body(A,5,24,step,S,index);break;
                case 2:
                        loop_body(A,13,9,step,S,index);break;
                case 3:
                        loop_body(A,10,16,step,S,index);break;
                case 4:
                        loop_body(A,11,15,step,S,index);break;
                case 5:
                        loop_body(A,12,9,step,S,index);break;
                case 6:
                        loop_body(A,2,27,step,S,index);break;
                case 7:
                        loop_body(A,7,15,step,S,index);break;
                case 8:
                        loop_body(A,14,6,step,S,index);break;
                case 9:
```

Figure A.2: Kernel component: md6_kernel.cu

```
                        loop_body(A,15,2,step,S,index);break;
                case 10:
                        loop_body(A,7,29,step,S,index);break;
                case 11:
                        loop_body(A,13,8,step,S,index);break;
                case 12:
                        loop_body(A,11,15,step,S,index);break;
                case 13:
                        loop_body(A,7,5,step,S,index);break;
                case 14:
                        loop_body(A,6,31,step,S,index);break;
                case 15:
                        loop_body(A,12,9,step,S,index);
                        N1+=16;
                        S = ((S << 1) ^ (S >> (W-1)) ^ (S & Smask));
                        break;
        }
        __syncthreads();

}


__device__
md6_word md6_byte_reverse( md6_word x ){
        x = (x << 32) | (x >> 32);
        return x;
}

__device__
void md6_reverse_little_endian( md6_word *x, int count ){
        int i;
        for (i=0;i<count;i++)
        x[i] = md6_byte_reverse(x[i]);
}

__device__
md6_control_word md6_make_control_word( int r,int L,int z,int p,int keylen, int d){
        md6_control_word V;
        V = ( (((md6_control_word) 0) << 60) | /* reserved, width  4 bits */
                (((md6_control_word) r) << 48) |           /* width 12 bits */
        (((md6_control_word) L) << 40) |           /* width  8 bits */
        (((md6_control_word) z) << 36) |           /* width  4 bits */
        (((md6_control_word) p) << 20) |           /* width 16 bits */
        (((md6_control_word) keylen) << 12 ) |     /* width  8 bits */
        (((md6_control_word) d)) );                /* width 12 bits */
        return V;
 }

/*
        Node ID's.
*/

__device__
md6_nodeID md6_make_nodeID( int ell,int i){
        md6_nodeID U;
        U = ( (((md6_nodeID) ell) << 56) |
        ((md6_nodeID) i) );
        return U;
}

/*
        Function: md6_pack

        Description: MD6 Preprocessing on the MD6 Buffer
*/

__device__
```

Figure A.2: Kernel component: md6_kernel.cu

```
void md6_pack( md6_word*N,int ell, int i,int r, int L, int z, int p, int keylen, int d,md6_word* B ){
        int j;  int ni,temp;  md6_nodeID U;  md6_control_word V; ni = 0;
        for (j=0;j<q;j++) N[ni++] = Q[j];       /* Q: Q in words     0--14 */
        for (j=0;j<k;j++) N[ni++] = 0;       /* K: key in words  15--22 */
        U = md6_make_nodeID(ell,i);             /* U: unique node ID in 23 */
        N[ni] = U;
        ni += u;
        V = md6_make_control_word(r,L,z,p,keylen,d);/* V: control word in   24 */
        N[ni]=V;
        ni += v;
        ni=25;
        temp = ni;
        for(j=ni;j<=87;j++){
                N[ni]=B[j-temp];
                ni++;
        }
}

/*
        Function: md6_compress_block

        Description: It invokes the MD6 compression routine on the designated MD6 buffer corresponding to
the Block ID
*/

__global__
void md6_compress_block(uint64_t *S1,md6_word *data,int levels,md6_word *N){
        int ell,z,p,keylen,d,L,r;
        int index;
        int nodes = blockIdx.x*levels;
        r=72;
        ell=(levels/4);
        z=p=keylen=d=L=0;
        index = nodes*64;
        md6_reverse_little_endian(&data[index],b);
        md6_pack(&N[index+41],ell,nodes,r,L,z,p,keylen,d,&data[index]);

}
__global__
void md6_compress(uint64_t *S1, md6_word *data,md6_word *N, int levels){
        int r=72;
        int nodes = blockIdx.x*levels;
        int index = (nodes*64);
        int j;
        /************* Execute the kernel **************/
        md6_main_compression_loop(&N[index],S1,r,c );
        for(j=25;j<=88;j++){
                data[index]=N[index+41+j];
                index++;
        }

}
__global__
void md6_writeback(md6_word *data, md6_word *N, int levels){
        int nodes = blockIdx.x*levels;
        int index = (nodes*64);
        int j;
        /********* Copy result from device to host ****************/

        for(j=25;j<=88;j++){
                data[index]=N[index+41+j];
                index++;
        }
}
__global__
void md6_rewrite(md6_word *data,int level){
        int buffers = 0;
```

Figure A.2: Kernel component: md6_kernel.cu

```
        int i,j,index;
        i=blockIdx.x*level*64;
        while(buffers<4){
                index=i+(buffers*64);
                j=0;
                for(j=0;j<=15;j++){
                        data[i]=data[index];
                        i++;index++;
                }
                buffers++;
        }
}
#endif // #ifndef _TEMPLATE_KERNEL_H_
```

Figure A.2: Kernel component: md6_kernel.cu