Faster File Matching Using GPGPUs

Deephan Mohan and John Cavazos

Department of Computer and Information Sciences, University of Delaware

Abstract

We address the problem of file matching by modifying the MD6 algorithm that is best suited to take advantage of GPU computing. MD6 is a cryptographic hash function that is tree-based and highly parallelizable. 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. In the parallel implementation, the MD6 program was partitioned and effectively parallelized across the GPU using CUDA. To demonstrate the performance of the CUDA version of MD6, we performed various experiments with inputs of different MD6 buffer sizes and varying file sizes. CUDA MD6 achieves real time speedup of more than 250X over the sequential version when executed on larger files. CUDA MD6 is a fast and effective solution for identifying similar files.

Keywords: CUDA, MD6, GPU computing, File matching

1. Introduction

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. The crux of the file matching application relies on the robustness and integrity of the hash generating algorithm. Various checksum generation algorithms like MD5[1], SHA-1[2], SHA-256[3], Tiger[4], Whirlpool[5], rolling hash have been utilized for file matching [6, 7]. The focus of this work is based on parallelizing **MD6** on the GPUs using the **CUDA** programming model. The **MD6** algorithm was ported to the GPUs and it was further deployed to solve the problem of file matching. The package that has been developed which eventually achieves the goal is **CUDA MD6**.

2. MD6 Algorithm

MD6 is the latest of the "message digest" algorithm and is ideally suited to exploit the parallelism presented by today's GPGPU architectures. Rivest et al.[8] describes the potential benefits of running the MD6 algorithm on parallel processors, and in particular on GPUs. Most of the 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. The design of MD6 is based on a highly parallelizable data-structure known as **"Merkle tree."** A tree-based design is the most natural approach for exploiting parallelism in a hash function. The computation proceeds from the leaves to 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. The structure of a Merkle tree is shown in Figure 1. In the MD6 implementation, the MD6 buffer size determines the number of levels in the tree and the bounds for parallelism.

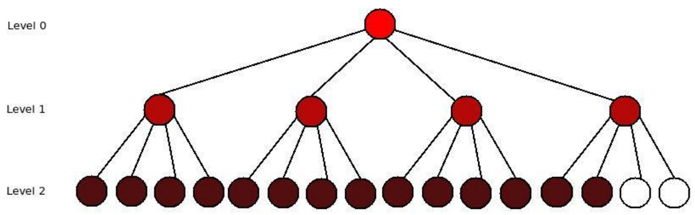

Figure 1: Structure of a Merkle tree: The diagram shows a Merkle tree with 2 levels. The computation proceeds from the bottom level to the top. Each node represents a data chunk. The hash computation of a node takes place when the sub-hashes of all the child nodes have been computed. If the number of 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).

2.1. MD6 Inputs

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. 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. K is a set of keys associated with the integrity of the hash function which is used for improving the security of the algorithm. Round r is a parameter that controls the number of times the MD6 compression function can be invoked on a given MD6 chunk.

2.2. *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. In this work, K is a nil set, since this work does not concern about security. B is the actual MD6 buffer which contains the data. U and V are auxillary inputs that are part of the compression mechanism. Each compression round performs 16 independent operations on each data block.

3. Related work

The NIST proposal for MD6 [8] describes the GPU implementation where the authors discuss the throughput based on hash lengths and number of cores. The results presented were entirely concerned with the GPU processing time of the kernel to showcase the parallelizing efficiency of MD6. Our work is based on the practical application of a GPU implementation of MD6 for file matching and evaluates the running time of the entire MD6 application, not just the kernel executing on the GPU. This distinction is important to make since it stresses the feasability of applying MD6 using GPU's for faster hash computation.

4. CUDA MD6

In MD6, each compression level and all of 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 round compressed one data block with 16 threads performing the 16 steps of the compression round. The number of CUDA blocks is limited to the size of the source file and finally to the physical memory of the GPU device.

4.1. *Description of Host Function*

The host function 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.

4.2. *Description of Kernel Functions*

The kernel contains the functions to perform the MD6 compression in a thread based design. Each kernel function has limited local memory access where local state variables can be defined. There are 3 kernel functions used in CUDA MD6: 1. A preprocessing module which performs the transformation from the input 64-word to 89-word. **(Grid size, Thread blocks) = (Number of data blocks, 1)**. 2. A compression module which performs the actual compression. **(Grid size, Thread blocks) = (Number of data blocks, 16)**. 3. A Hash aggregation module stores the sub-hashes of each thread block at the appropriate indices for the next level. **(Grid size, Thread blocks) = ((Number of data blocks/4), 1)**.
The kernel call invoking the compression module performing one round of MD6 compression along with the compression routine is shown in Figure 2.

```
/* Invoke md6 compression routine on each MD6 buffer */
dim3 grid(num_blocks,1,1);
dim3  threads( 16, 1, 1);
md6_compress<<<grid,threads>>>(S1,d_idata,N,(leaves/4),num_blocks);
cudaThreadSynchronize();

/* One round of MD6 compression call */
int tx = threadIdx.x % num_threads;
int ty = threadIdx.y;
step = tx+ty;
index = 89;
loop_body(A,rsArgs[tx],lsArgs[tx],step,S,index);
        if (tx == 15) //Last thread
        {
                N+=16; //Move to the next round
                S = ((S << 1) ^ (S >> (W-1)) ^ (S & Smask));
        }
__syncthreads();

/* MD6 compression routine */
__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            */
}
```

Figure 2: Figure shows md6 compression kernel invocation and one round of md6 compression routine.

5. CUDA MD6 for File matching

CUDA MD6 can perform file matching in two modes:

- Direct hashing for individual files
- Recursive hashing for archive of files

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. In case of working with larger files, CUDA MD6 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.

6. Experimental setup

The CUDA experiments were conducted on Linux machines equipped with Nvidia GeForce 8800 GTX card (112 cores) and quad-core Intel Xeon E5335 CPU. The CUDA MD6 was developed using CUDA toolkit version 2.2.

7. Discussion of results

Figure 3 shows the CUDA experiment performed with different buffer sizes. The speedup follows a linear trend with the increase in the size of the buffers which is explained by the fact that the size of the Merkle-tree shrinks by a factor of N, when the MD6 buffer size is increased by N. For each layer of the tree, the number of thread blocks required to perform the compression functions decreases with the increase in size of the buffer. Due to lesser number of kernel calls, the GPU processing time is faster for smaller buffer sizes. Figure 4 shows the speedup experiment conducted on individual files of sizes - 25MB, 50MB, 100MB, 500MB and 1GB respectively. The size of the MD6 buffer was 512 bytes and the number of compression rounds was assigned a default value of 72. The number of compression rounds determines the integrity of the compression function. To test the speedup of the parallel MD6 over the serial version, the program was executed on a range of files in the increasing order of sizes. The speedup analysis was done after the execution time of serial and parallel MD6 was averaged over 10 runs. The speedup increase is geometric in nature. For smaller files, 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 and host-device data transfer overhead. It is also evident from Figure 4 that significant speedup is only achieved when the input file size exceeds a certain threshold. This file size threshold determines the point below which the MD6 hashing can be done on the CPU instead of engaging the GPU. This file threshold for the given buffer size was found to be approximately 25MB. The speedup of the experiment is determined by a number of factors like buffer size, number of compression rounds and thread block size. Increasing the number of compression rounds increases the speedup since the number of independent compression operations becomes higher. The startup time for the device (Initialization) is almost constant when CUDA MD6 is run on files of any size. This initialization delay also affects the real speedup of the application. A hybrid approach incorporating the texture memory can be employed to get a higher speedup even for smaller files.

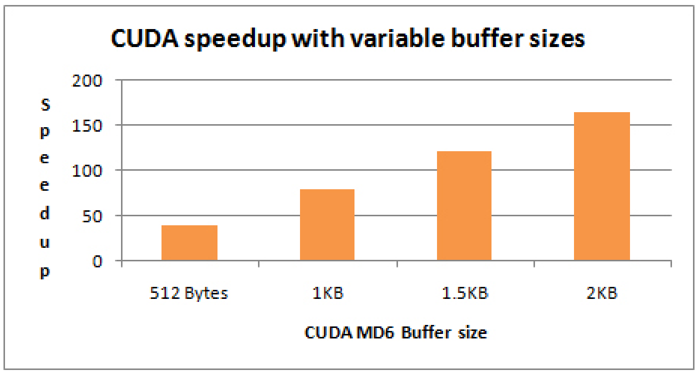

Figure 3: Real time speedup of CUDA MD6 with varying MD6 buffer sizes: CUDA MD6 was executed with varying CUDA buffer sizes of 512 Bytes, 1KB, 1.5KB and 2KB on a 500MB source file.

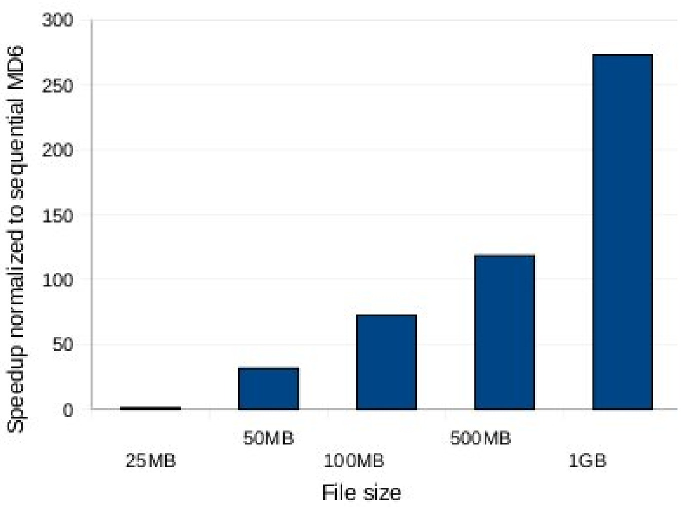

Figure 4: CUDA speedup experiment on individual files with a data buffer size of 512KB. We achieve an identical execution time with that of sequential MD6 for a file size of 25MB up to a speedup of 250X for a file of size 1GB.

8. Conclusion

In this work we implemented a file matching application for identifying similar files. The hashing algorithm used for the design of this application is MD6 that is very well suited for portability on the GPUs. The MD6 compression function which performs the brunt of computation is highly parallelizable and it was chosen as the kernel function. The MD6 algorithm was mapped to the GPU, by intelligently designing the partition procedure and also exploiting the multi-core architecture of the GPU's. The file matching experiments conducted reveal that CUDA MD6 can achieve real time speedup up to 250X, when executed on larger files. Hence, we conclude that CUDA MD6 is one of the solutions for faster file matching on a large scale.

References

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

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

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

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

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

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

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

[8] Ronald L.Rivest. *"The MD6 hash function. A proposal NIST for SHA-3."* URL:http://groups.csail.mit.edu/cis/md6/revision-2009-1-15/Supporting_Documentation/md6_report.pdf Oct.2008. (Accessed: 14 May. 2010)