Parallel and High Speed Hashing in GPU for Telemedicine Applications

Wai-Kong Lee, *Member, IEEE,* Raphael C.-W. Phan, *Member, IEEE,* Bok-Min Goi, *Senior Member, IEEE,* Lanxiang Chen, *Member, IEEE,* Xiujun Zhang, *Member, IEEE,* Naixue Xiong, *Senior Member, IEEE*

***Abstract*—: With the advent of telemedicine technology, many medical services can be provided remotely, which greatly enhances the welfare of our mankind. However, security and privacy of medical data transmitted through telecommunication systems remain a serious issue to be resolved when deploying such services. In particular, the medical images and data stored in the cloud or transmitted over insecure channel, may suffer from unauthorized modifications by malicious attackers. Hence, integrity of such medical data is of utmost importance for telemedicine applications. Cryptographic hash functions (e.g. SHA-3) can be used to ensure the integrity of medical data communicated over insecure channel. However, when the volume and size of medical data grows (e.g. high resolution medical image), it is difficult for conventional CPU-based system to hash these data in timely manner. In view of that, we are motivated to research on improved implementation techniques of Keccak hash function in massively parallel platforms, as the result of such work can be used in improving the speed performance of telemedicine applications. Graphical processing unit (GPU) is one of the emerging platforms with massively parallel processing power that can be harnessed to solve computational problems much faster than conventional CPUs. In this paper, we present the efficient implementation of tree-mode Keccak-*f(1600)* in GPU and investigate the effect of parallel granularities by hashing one copy of Keccak permutation function using 1 thread, 5 threads and 25 threads respectively. We also proposed a new technique to implement tree-mode Keccak-*f(1600)* based on Dynamic Parallelism offered in new NVIDIA GPU. Our experimental results show that parallel granularity of one thread produces the highest hash throughput at 28.51 Gbps. The high hash rate of such implementation can greatly enhance the integrity check for medical data in telemedicine applications.**

***Index Terms*—Security, Telemedicine, GPU, SHA-3**

I. Introduction

Telemedicine is able to enhance the quality of life for mankind, as medical data can be communicated remotely through high speed Internet connections [1], [2]. Patients can now receive professional treatments or medical services from medical experts from distance, which is especially crucial for people living in remote area or facing emergency situations. However, security concern is an important facet of telemedicne which receives a lot of attentions recently, due to the privacy concern from patients. One of the most important security aspects is the integrity of medical data, which can be protected through cryptographic hash function.

Considering some well deployed cryptographic hash functions like MD5 and SHA-1 are under threat in recent years due to advancement in cryptanalysis [3]–[5], the US National Institute of Standards and Technology (NIST) started a new public competition in 2007 to select a SHA-3 algorithm for standardization. The competition started with 64 first round candidates, with 14 of them advancing to the second round. The five finalists that remained after that were namely Keccak [6], BLAKE [7], JH [9], Skein [8] and Grøstl [10]. In October 2012, Keccak was selected as the new SHA-3 standard. Since Keccak is able to provide high security properties and is likely to be adopted by the industry in near future, we have selected Keccak to protect the integrity of medical data used in telemedicine applications.

Although hash function like Keccak can be used to check the integrity of medical data to prevent malicious modifications, adopting it in telemedicine applications may not be straightforward, due to the sheer volume and size of the data involved. For example, transmission of high resolution medical images and videos are often required to provide medical services to the patients. In such cases, conventional CPU-based systems may not be able to complete the protection (hashing the original data) and integrity check (hashing the received data) in timely manner. This may deteriorate the user experience (affecting the quality of service [38]) as well as causing delay in curing the patients. This motivates us to research on the fast implementation techniques for Keccak in Graphical Processing Unit (GPU), which is massively parallel platform with high computational capability.

Keccak is a hash function based on the sponge construction, which is inherently a sequential process. Although the internal permutation function can be implemented in parallel, this alone is still insufficient to harness the massively parallel computing power of GPUs. More aggressive design requires the hash function to be implemented in tree structure [11], [12]. The authors of Keccak outlined two approaches for implementing tree-mode Keccak [13]–[16], namely Final Node Growing (FNG) and Leaf Interleaving (LI). FNG mode allows the tree nodes to grow with increasing input file size, while LI mode

Wai-Kong Lee and Bok-Min Goi are with Centre of Cyber Security, Universiti Tunku Abdul Rahman, Malaysia.

Raphaël C.-W Phan is with Faculty of Engineering, Multimedia Universiti, Malaysia.

Lanxiang Chen is with College of Mathematics and Informatics, Fujian Normal University, Fujian Provincial Key Laboratory of Network Security and Cryptology, Fuzhou, China.

Xiujun Zhang is with School of Information Science and Engineering, Chengdu University, Chengdu, China.

Naixue Xiong is with School of Computer Science and Technology, Tianjin University, and Department of Mathematics and Computer Science, Northeastern State University.

Manuscript received

Corresponding authors: Lanxiang Chen (e-mail: lxiangchen@fjnu.edu.cn), Xiujun Zhang (e-mail: woodszhang@cdu.edu.cn), and Naixue Xiong (e-mail: xiongnaixue@gmail.com).

has a fixed tree structure. Considering that GPU has finite memory resources, it is more appropriate to implement a hash function that has a fixed tree structure, i.e. LI mode in this case. For the rest of this paper, we will focus on LI tree-mode Keccak and present our implementation design techniques for this.

Although general purpose microprocessors can naturally be used for implementing cryptographic algorithms, dedicated hardware implementation remains attractive because cryptographic algorithms involve operations which are poorly supported by general purpose processors. FPGA and ASIC are popular hardware platforms to implement advanced cryptographic algorithms (e.g. AES & ECC [23]). For instance, Koziel *et al.* [25] proposed a hardware architecture to accelerate isogeny-based cryptography, which is a candidate for post-quantum cryptography. On the other hand, Dai *et al.* [36] presented an FFT based exponentiation hardware architecture for RSA. Kerckhof *et al.* [27] presented the compact implementation in FPGA of five finalists of the SHA-3 competition. Although hardware implementation enjoys performance boosts compared to general purpose processors, it also suffers from a few drawbacks. Notably, hardware implementation is inflexible, difficult for subsequent upgrade and maintenance. Hardware implementation like ASIC often involves expensive fabrication cost and requires specialized design skill which is in turn causing longer development period.

GPUs have emerged as one of the most promising platforms for scientific computing. Since the introduction of general purpose API for programming graphic processors (e.g. CUDA from NVIDIA and Stream from ATI), GPU is widely used in scientific simulations [28], [29], model buildings and algorithm implementations. A GPU consists of multiple Streaming Multi-processors (SM); each SM consists of many cores. A GPU typically consists of tens to hundreds of cores (GTX780 from NVIDIA has 12 SMs with a total of 2304 cores). With many cores operating concurrently, GPU is an ideal candidate for applications that need to execute relatively simple programs on many data. GPU is also widely used in accelerating the implementation of advanced cryptographic algorithms in recent years [30]–[36].

In general, the basic idea to keep in mind is that the GPU is used as a co-processor executing parallelizable codes, while the CPU will handle sequential tasks and management. Careful design therefore on the GPU algorithm, data transfer between CPU and GPU, and smart usage of various GPU memories are needed to obtain high performance. This indicates that the gist of a successful implementation should consider all major components in the heterogeneous platform instead of focusing on the GPU alone. Detailed programming model for this heterogeneous platform will be discussed in Section 3.

There are five other GPU implementations of Keccak available in the literature [18]–[22]. Bos *et al.* [18] implemented the Leaf Interleaving (LI) tree-mode Keccak-256, based on GTX295 with 30 SMs (GTX295). Since its source code is closed, we only use their result as a benchmark. The work done by Sevestre [19] also implements LI tree-mode Keccak with the heterogeneous programming model, but it is based on Keccak-*f(800)* which uses less memory and has smaller value of r and c compare to ours. Since capacity c is the security parameter, reducing this parameter also reduces the security level.

Keccak-*f(1600)* has 25 internal states, each of the state can be represented by a 64-bit word. Chindemi *et al.* [20] implemented Keccak-*f(1600)* by using 25 threads to compute 25 Keccak internal states concurrently, but they do not implement tree-mode Keccak. Cayrel *et al.* [21] is the most relevant related work. They explored the idea to construct a LI tree-mode Keccak with variable height (H=0 to H=4). The basic kernel uses 25 threads to compute 25 Keccak internal states concurrently. By using shared memory and some lookup tables stored in GPU constant memory, they manage to implement a compact kernel with a minimum for loops. However, the main drawback is that their work suffers from bank conflicts due to the memory access pattern of the Keccak algorithm, which was an open problem highlighted by Cayrel *et al.* [21] in their paper. Lowden et al. [22] evaluated various ways to optimize the tree-mode Keccak implementation in GPU, but they did not explore the Dynamic Parallelism feature to improve the speed performance.

The straightforward way to construct a tree-mode hash function is to slice input data into multiple copies, and run multiple threads concurrently with each thread hashing a copy of the sliced data. For certain hash functions, a copy of sliced data can be actually hashed by multiple threads instead of a single thread; Keccak is one of the hash functions that falls into this category. In this paper, we focus on investigating the effect of parallel granularity of Keccak permutation function in GPU platform. Specifically, we implemented the three plausible versions specific to the Keccak permutation function, namely single thread version (we call it 1T-Keccak), five threads version (5T-Keccak) and 25 threads version (25T-Keccak) on our GPU system. The choice of 25 threads is due to the fact that Keccak-*f(1600)* state can be represented by 25 lanes of 64-bit words, each lane can be hashed simultaneously. On the other hand, 5T-Keccak hashes a plane or a sheet of Keccak state using one thread, so five threads can hash the entire Keccak state simultaneously. We discuss on the performance yield by these three parallel granularities and the design considerations that should be taken care of.

The conventional way to implement tree-mode hash function is to launch a kernel for each level of tree height sequentially, and the upper tree level needs to wait for the lower tree level to complete before it can progress. This process required explicit control from CPU. In order to address this limitation, we proposed a better approach by utilizing Dynamic Parallelism feature in recent NVIDIA GPU to manage the kernel launches of each tree level. With this approach, we offload the kernel launch management task to GPU and free up the CPU computing resources for other tasks. This is especially important for server applications as the CPU cores are usually busy in serving various requests from the network

Towards our aims, we also propose specific optimization techniques for Keccak implementation on GPU. These include asynchronous memory copy, overlapping CPU and GPU execution, configuration setting to avoid shared memory bank conflicts, data pre-fetch and loop optimizations (unroll and

TABLE I
PERMUTATION FUNCTION OF KECCAK-*f[b]*

Round[b](*A*,RC)	
θ STEP	
$C[x] = A[x, 0] \oplus A[x, 1]$	$\forall x$ in 0...4
$\oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$,	
$D[x] = C[x - 1] \oplus \text{ROT}(C[x + 1], 1)$,	$\forall x$ in 0...4
$A[x, y] = A[x, y] \oplus D[x]$,	$\forall (x, y)$ in (0...4, 0...4)
ρ AND π STEPS	
$B[y, 2x + 3y] = \text{ROT}(A[x, y], r[x, y])$,	$\forall (x, y)$ in (0...4, 0...4)
χ STEP	
$A[x, y] = B[x, y] \oplus ((\text{NOT } B[x + 1, y])$ AND $B[x + 2, y])$,	$\forall (x, y)$ in (0...4, 0...4)
ι STEP	
$A[0, 0] = A[0, 0] \oplus \text{RC}$	
return *A*	

A[x,y] denotes particular lane in that satte, B, C and D are intermediate variables. RC is the round constants and r[x,y] is the rotate offset [13].

inversion). The organization of this manuscript is presented below. Firstly, we give an overview to the Keccak hash function and its tree structure for parallel implementation in Section II. Then, we introduce the hardware architecture and programming model for GPU in Section III, follow by the details of GPU implementation in Section IV. After that, the experimental results are presented in Section V, and the conclusions is presented in Section VI.

II. KECCAK HASH FUNCTION AND TREE BASED STRUCTURES

Keccak is constructed based on sponge construction with seven modes, indicated by Keccak-*f[b]*, where $b = 25 \text{ x } 2^l$ and l can be 0–7. The state in Keccak permutation is organized as an array of 5×5 lanes of length $w \in \{1, 2, 4, 8, 16, 32, 64\}$. For a platform with 64-bit, the permutation state b can be expressed as 25 x 64 bit word; hence, the permutation state can also be expressed in the form of $b = 25w$ where w represents the permutation width in specific platform.

The permutation f is applied repeatedly to the state $b = r + c$ with fixed length, where r is the bit rate and c is the capacity. Bit rate r determines the implementation speed while c determines the security strength. Inside Keccak-*f[b]* is a sequence of same round operations, where the total number of round $N_r = 12 + 2l1$. Each round of permutation consists of five steps, namely θ, ρ, π, χ and ι, shown in Table I.

We implemented the default Keccak mode, Keccak-*f[1600]* with $r = 1024$ and $c = 576$. The recommended number of round for Keccak-*f[1600]* is 24 and the permutation width w is 64 bit.

Keccak-*f[b]* is an iterated process that is divided into two phases: Absorb and Squeeze. The process is briefly described below:

i) Initialize the state to 0.
ii) Pad the input message with multi-rate padding rule (pad10*1) [13]
iii) In the absorbing phase, input data blocks are sent into the permutation function iteratively. The permutation state is first XORed with r-bits input data, followed by the permutation process.
iv) The squeezing phase takes place after all data blocks are absorbed. The first r-bits of the state from the absorbing phase are returned as the output block. If the user requires more output blocks of r-bits, permutation is applied to the state repeatedly to provide the next output block of r-bits.

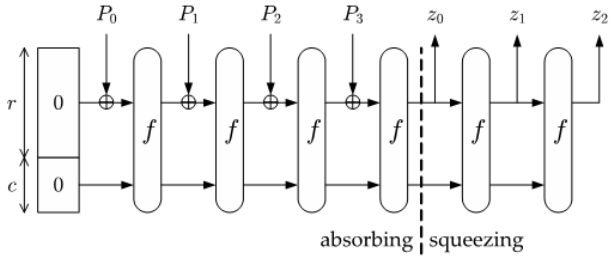


Fig. 1. The Sponge Construction [15].

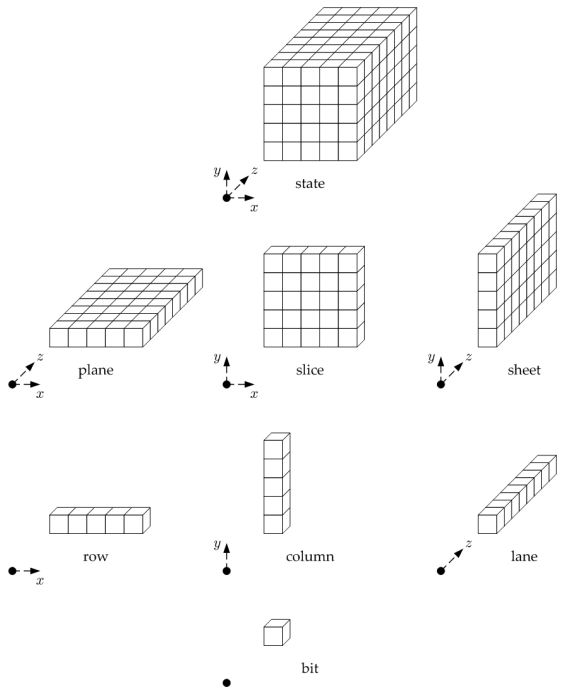


Fig. 2. Naming Convention for the Parts of Keccak-f State [13].

Figure 2 shows the naming convention for the parts of the Keccak internal state. In a 64-bit system, the internal state for Keccak-*f(1600)* can be naturally represented by 25 words, each word represent a lane. This shows possibility for internal parallelism in the Keccak permutation function. Besides, it is also possible to run Keccak with five processes running concurrently, each hashing a plane or a sheet.

Keccak hash function is naturally a sequential process, hence it is difficult to implement this directly into GPU and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2849439, IEEE Access

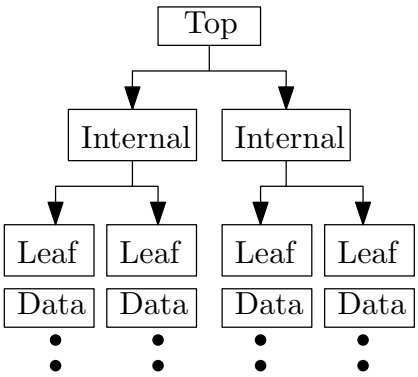


Fig. 3. Leaf Interleaving Tree Structure for Keccak-*f*.

expect performance speed up from it. One of the possible ways to utilize parallel execution in GPU is to use tree-mode implementation. In [13], G. Bertoni *et al.* proposed two ways to implement tree-mode Keccak: Final Node Growing (FNG) and Leave Interleaving [LI]. For FNG, the number of leaves and degree of the top node grow as a function of input data size. For LI, the structure of the tree is fixed, but the input data are interleaved (hashed serially) into the leafs. Figure 3 shows an example of LI tree structure for Keccak, with height H=2 and degree D=2. For a detail explanation on the tree structure for Keccak, please refer to [13].

III. Overview of The Target Platform

This section describes the main points about the GPU platform, in particular its programming model, memory hierarchy and the architecture of the specific GPU we run on. A proper understanding of these matters is crucial to ensuring an optimized implementation strategy for tree-mode Keccak on GPUs.

A. CUDA Heterogeneous Programming Model

Compute Unified Device Architecture (CUDA) is the software technology developed by NVIDIA to allow programmers to utilize the GPU for non-graphic purposes of computations. The CUDA API reduces effort to program the GPU for general purposes with extension to C and FORTRAN programming language. However, a deeper knowledge of the GPU's architecture, particularly memory, threads andblocks, is crucial in order to harness its great computational power.

Besides C and FORTRAN, CUDA provides the user the flexibility to code in low level Parallel Thread Execution (PTX) language. PTX provides an instruction set for general purpose parallel programming, which is regarded as "pseudo-assembly code" for NVIDIA GPU. It also aims to provide a machine-independent instruction set architecture (ISA) for C/C++ and other compilers to target. PTX instructions can be added into standard C/C++ program via inline assembly.

GPUs can execute many threads in parallel; each thread will execute the same instructions on different data sets. The thread level codes that a programmer writes are called the kernel. Each streaming multiprocessor (SM) within a GPU partitions every 32 threads into a warp. All 32 threads in a warp execute the same instruction at the same time; as a result, full efficiency is realized when all 32 threads of a warp have same execution path. Branch divergence will seriously degrade the performance; hence it should be avoided if possible. The warp scheduler schedules as many warps as possible in order to hide any memory access or instruction latency. So it is important to maintain a large active thread pool to achieve high occupancy and keep all the warps busy. Multiple threads form a block, multiple blocks then form a grid. Figure 4 shows the relationship between grid, block and thread inside a SM. There is a maximum limit for threads per block and number of blocks per SM, depending on its Compute Capability. For example, the GTX780 GPU with Compute Capability of 3.5, can house a maximum of 16 thread blocks and maximum 2048 threads per SM.

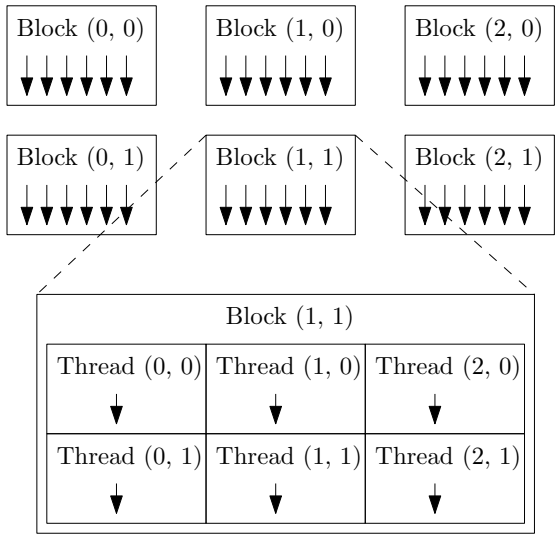


Fig. 4. Relationship between Grid, Block and Thread inside a SM.

CUDA assumes that the CPU and GPU have their own memory space, referred to as host memory and device memory respectively. A typical CUDA program will follow the steps below:

i) Allocate and initialize host and device memory.
ii) Copy input data from host memory to device memory.
iii) Launch kernel for computation. The pointer for the device memory and some other parameters are passed to the kernel. In the meantime, control is passed back to the CPU even though the kernel is still being computed on the GPU. Store the final data from the kernel computation in device memory.
iv) While waiting for the GPU to complete its execution, CPU can perform other tasks.
v) Copy data from device memory back to the host when all GPU executions are completed.

More advanced GPU programming model involves the use of streams [17]. A stream is a sequence of commands that execute in order. Different streams may execute their commands out of order with respect to one another or concurrently. Since NVIDIA GPU has separate copy and kernel engines, a stream can be used to overlap the process of memory copy and kernel

execution. Some GPU devices with two copy engines can even overlap memory copy from host to device and device to host. Devices with Compute Capability 2.0 and above are capable of running multiple copies of kernels concurrently, which greatly improves the parallelism if it is used together with streams. However, the use of streams is limited by several factors like data dependency and instructions issue order.

B. Memory Hierachy

Global memory is the largest off-chip memory in the GPU, but it is also the slowest. It is used to store the data transferred from the host, accessible by all threads in all SM. Global memory needs to be accessed in coalesced manner (128 bytes), or else it will suffer great performance degradation.

Constant memory is cached memory that allows the user to store read-only data. It is an ideal choice to store and broadcast read only data to all threads on the GPU.

Texture memory is bound to global memory and provides cache functionality. It is optimized for 2D spatial access patterns.

Shared memory is accessible by all threads within the same thread block. It is commonly used to hold temporal data so that threads within the same block can cooperate. Shared memory is organized in banks that are 32-bits. If multiple requests are made by different threads to the same address or to different addresses in the same bank, bank conflicts will occur. Bank conflicts will seriously degrade performance as the memory access is serialized now. However, if it is designed carefully to avoid bank conflicts, shared memory can provide very fast access speed.

Registers are the fastest memory in GPU, and only accessible locally by each thread. Latest NVIDIA GPU with Compute Capability 3.5 have 64KB registers per SM, and maximum 255 registers per thread. Since registers are the most precious resource in GPU that enable us to deliver peak performance, they should be used carefully. Register use can affect the maximum threads that can run simultaneously. For example, if the SM is running 2048 threads, only 32 registers can be used. If a kernel uses more registers than its maximally allowed limit, the compiler will spill extra register usage into "local memory". CUDA API does not allow programmers to have explicit control on which variables to reside in the register, it is determined by the compiler. Even PTX itself is not a real assembly language; it is just an intermediate description. To the best of our knowledge, the only way to fully control register allocation process is to develop a new assembler [10].

Local memory resides in global memory, but it is cached at L1 cache. Register spilling effect is determined by the compiler; the programmer does not have explicit control over this aspect.

C. GTX780 and GTX295

In this paper, we have chosen two platforms for implementing the Keccak hash function. GTX780 was chosen because it has the Dynamic Parallelism feature that is useful to reduce the time to manage the hash tree; this technique is applicable to all future GPU that supports Dynamic Parallelism. On the other hand, GTX295 was used to provide a fair benchmark against earlier implementation. GTX780 is a device with Compute Capability 3.5. It has 12 SMs, each of the SMs consists of 192 cores, running at 900Mhz. It is equipped with 3GB global memory, 64KB register file per SM, configurable L1 and shared memory (total 64KB). The shared memory and L1 cache can be configured in four ways:

i) 16KB Shared Memory, 48KB L1 cache
ii) 48KB Shared Memory, 16KB L1 cache
iii) 32KB Shared Memory, 32KB L1 cache
iv) No preference (default)

Shared memory can also be configured to 64-bits addressing mode. With this addressing mode, a shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 64-bit word (even though the addresses of the two sub-words fall in the same bank). This feature is very useful for Keccak-*f(1600)* implementation as its internal state is 64-bit wide. By configuring shared memory to this addressing mode, bank conflicts can be minimized [17].

GTX295 is a device with Compute Capability 1.3. It has 30 SMs, each of the SMs consists of 16 cores, running at 1242Mhz. It is equipped with 1.792GB global memory, 16KB register file per SM and 16KB shared memory. The shared memory size and addressing mode in this device is not configurable.

IV. Keccak Implementation Design on GPU

We implemented LI tree-mode Keccak-*f(1600)* with r=1024 and c=576 on GPU platform, based on the techniques discussed in the following subsections.

A. Parallel Granularity

We implemented the three plausible versions of LI tree-mode to investigate the effect of parallel granularity in Keccak permutation function.

1) 1-Thread Keccak (1T-Keccak): In this mode, one thread is used to hash a copy of Keccak. The kernel is completely unrolled to minimize the use of for loops and lookup tables. The only lookup table used is the round constant, which is stored in constant memory. Since no thread cooperation is needed, no shared memory is used. This implementation has an advantage that it does not need any synchronization and data sharing between threads. The entire permutation process take place in a thread, hence no parallelism occurred within the thread. This granularity only utilizes parallelism in the tree structure.

2) 5-Thread Keccak (5T-Keccak): In this mode, five threads cooperatively hash a copy of Keccak, with each thread hashing a plane or a sheet. We implemented both the plane and sheet version of 5T-Keccak. This implementation need shared memory to share intermediate state values and variables across multiple threads. The calculation in Keccak is based on modulo-5, which is an expensive operation. Hence, this kernel uses lookup tables to avoid computing expensive modulo calculations on the fly. These tables are stored in constant memory. This granularity utilizes both parallelism within

the Keccak permutation function and parallelism in the tree structure.

3) 25-Thread Keccak (25T-Keccak): In this mode, 25 threads cooperatively hash a copy of Keccak, with each thread hashing a lane of the internal state. This implementation also needs to use shared memory and lookup tables for the same reason as 5T-Keccak. The NVIDIA warp scheduler will always group 32 threads into a warp, and all threads within the same warp must run common instructions at a time to avoid warp divergence. As a result, we need to launch 32 threads for this kernel to avoid warp divergence. 25 threads will be doing the actual work of hashing while seven other threads will be idle. This granularity utilizes both parallelism within Keccak permutation function and parallelism in the tree structure.

B. Kernel Launch Management

Recent NVIDIA GPU with Compute Capability 3.5 offer an advanced feature named Dynamic Parallelism, whereby the GPU kernel can launch another kernel by itself. In conventional GPU, the kernel can only be launched by CPU, so the algorithms that need multiple kernels to complete require full control from CPU to manage the kernel launches. With Dynamic Parallelism, the CPU only needs to launch the kernel once, then this GPU kernel can manage subsequence kernel launches within GPU, which eventually free up CPU resources for other tasks. Dynamic Parallelism also benefit algorithms that require recursive function call (e.g. quick sort). Figure 5 and 6 illustrate how this advanced feature works.

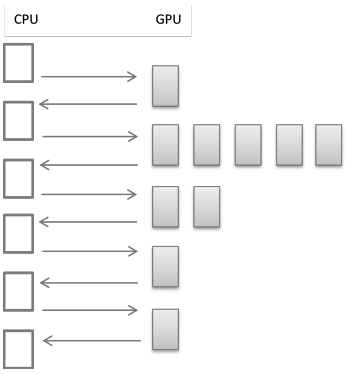


Fig. 5. Without Dynamic Parallelsim

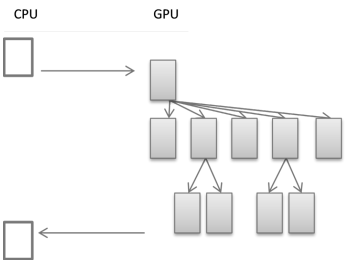


Fig. 6. With Dynamic Parallelsim

We exploited this feature by launching a manager kernel from CPU, and let this manager kernel keep track the execution of each Keccak tree levels. When a tree level complete its execution, the manager kernel will launch the next tree level and this process will continue until it reached the top tree level. With this approach, the CPU do not involve explicitly in controlling the kernel launch at each tree levels; hence it is freed up to handle other tasks.

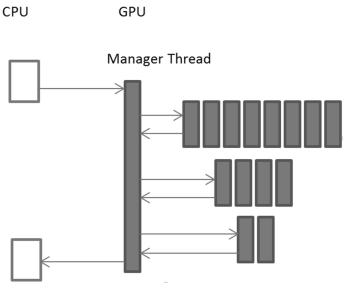


Fig. 7. Example of Tree-based Keccak Implementation With Dynamic Parallelism (H=3)

C. Prefetch Data

During the absorbing phase, the input data are sent to the Keccak permutation function and XORed with the current internal state. The conventional way to perform this is as below:

1: **for** $i \leftarrow 0\ to\ (r/w) - 1$ **do**
2: $\quad state[i] \leftarrow state[i] \oplus data[i]$
3: **end for**

In NVIDIA GPU, arithmetic instructions and memory load/store instructions can be executed concurrently, as long as there is no dependency between the executing instruction and data being load/store. To utilize this feature, we prefetch the input data before XORing it into the state, so that address calculation and bit-wise XOR operation can run in parallel with the memory copy operations. The generic syntax is as below:

1: **for** $i \leftarrow 0\ to\ (r/w) - 1$ **do**
2: $\quad temp\_var \leftarrow prefetch\_data$
3: $\quad data \leftarrow data + 1$
4: $\quad prefetch\_data \leftarrow data[i]$
5: $\quad state[i] \leftarrow data[i] \oplus temp\_var$
6: **end for**

Line two is the step to copy prefetched data into a temporary variable. Line three perform address calculation concurrently with previous instruction. Line four prefetch next data item into the variable prefetch_data. Line five describe the step to XOR current data into Keccak state concurrently while prefetching next data.

D. Loop Optimization

We also apply two loop optimization techniques in the Keccak permutation function. For 1T-Keccak, we manually unroll the entire kernel. For 5T-Keccak and 25T-Keccak, we utilize the loop inversion technique by replacing a while loop with if block containing a do...while loop to reduce jump

instruction, as jump instructions by nature introduce pipeline stalls.

The benefit of using loop inversion is illustrated below. Consider the execution of a while loop with 10 iterations, it will execute 11 jump instructions (GOTO) before escaping the loop:

```
1: i ← 0
2: B1:
3: if i ≥ 10 then GOTO B2:
4: end if
5:     a[i] ← 0
6:     i ← i + 1
7:     GOTO B1:
8: B2:
```

However, with loop inversion the jump instruction is reduced to only 9.

```
1: i ← 0
2: if i ≥ 10 then GOTO B2:
3: end if
4: B1:
5:     a[i] ← 0
6:     i ← i + 1
7: if i < 10 then GOTO B1:
8: end if
9: B2:
```

E. Avoiding Shared Memory Bank Conflict

For 5T-Keccak and 25T-Keccak, we use shared memory to store internal state and temporary variables. The variables we used to store internal state are 64-bit wide, which means that access to shared memory should be done in 64-bit as well. In GPU with Compute Capability lesser than 3.0, a 64-bit data access is done in two separate 32-bit accesses, which increase the chances for multiple threads to access the same memory bank. This in turn creates high chances for bank conflicts to occur and slow down the memory access performance.

The target platform we use (GTX780) is a device with Compute Capability 3.5; it offers a useful feature to configure the shared memory to 64-bits addressing mode. By doing this, the 64-bit access to shared memory is done with only single access, which in turn avoids bank conflict. Our implementation adopted this configuration and able to eliminate the bank conflict problem that Cayrel *et al.* [21] are facing. For another target platform GTX295, it does not allow user to configure shared memory addressing mode, so we are not able to apply this technique.

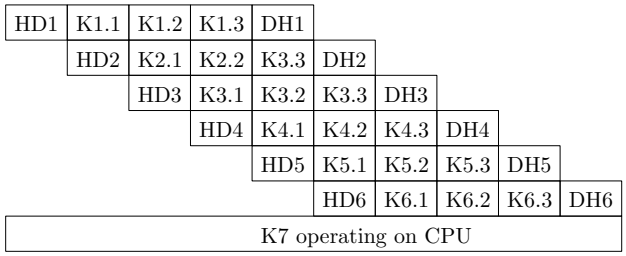
F. Concurrent Execution

CUDA not only provides thread level parallelism, it also allows multiple streams of kernels to run concurrently [24], and it can overlap memory copy with kernel and CPU execution. To illustrate this idea, we refer to Figure 8 and Figure 9.

cudaMemcpyAsync(H2D)	kernel <<<>>>	cudaMemcpyAsync(D2H)

Fig. 8. Serial execution with only one stream.

Figure 8 shows a typical CUDA program with only one stream, where memory copy from host to device, kernel execution and memory copy from device to host are executed serially.



HD: Host to device

DH: Device to host

K: Kernel broken into multiple parts

Fig. 9. Concurrent execution with multiple streams and CPU.

Figure 9 shows an example where the memory copy and kernel execution are overlapped, with the CPU doing other tasks concurrently. In this example, the kernel is divided into three parts (Kx.1 to Kx.3), while memory copy from host to device (HD1 to HD7) and device to host (DH1 to DH7) are divided into six parts to further improve the overlapping effect.

In order to utilize this programming model, we break the input data into multiple sections (depending on how many streams we use), each of the streams will hash one section of the data. We only apply this to the Keccak tree leaf (bottom level of tree) as it is the most time consuming process. Kernels for internal tree level are launched consecutively after the tree leaf kernels complete execution. The top root level is hashed by the CPU.

V. Experimental Results and Discussions.

We implemented LI tree-mode Keccak based on the three granularities and optimization techniques detailed in Section IV. We executed the experiments in a workstation system comprising an Eight Cores 4 GHz CPU, 16 GB of RAM, CUDA SDK 5.5, GTX780 with Compute Capability 3.5 and GTX295 with compute capability 1.3. The first experiment examine the effect of tree heights (varies from H=1 to H=7) to the hash throughput in GTX780 and GTX295, while the second experiment examine the effect of various input data size (range from 4KB to 256MB) to the hash througput in same platform. For a fair comparison, we did not use Dynamic Parallelism in this experiment setting for GTX780, because GTX295 does not support this feature. The range of input size was chosen to cover small file (KB) and large size (MB).

The tree degree is fixed at D=4 so that we can perform a direct comparison with the work done by Cayrel *et al.* [21]. The main difference between our work and Cayrel *et al.* [21]. is that we configured the shared memory to operate on 64-bit addressing mode to avoid bank conflict. We do not implement H=0 as it is equivalent to hashing in serial form. Hash throughput is almost stable when tree height H≥6, so our experiment stops at H=7. For 25T-Keccak, 25 threads hash

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2849439, IEEE Access

one leaf; for 5T-Keccak, five threads hash one leaf, while 1T-Keccak uses one thread to hash one leaf. For 5T-Keccak, we only present the results of sheet version, as the plane version also shows very similar results.

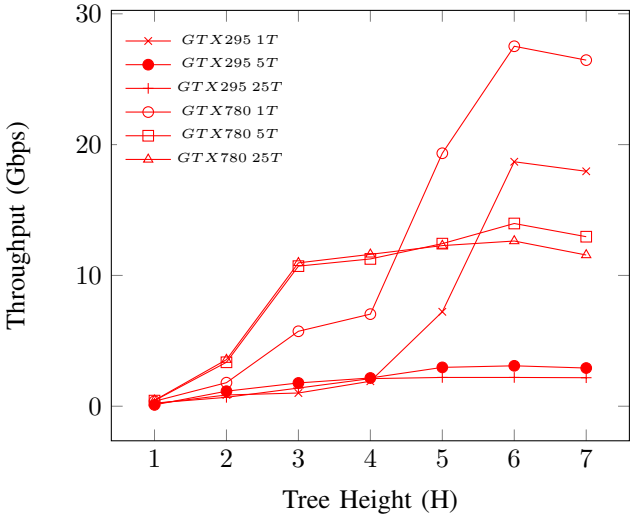


Fig. 10. Throughput in GTX295 and GTX780, with varying tree height. The work in GTX780 was implemented without Dynamic Parallelism.

Figure 10 shows that 5T-Keccak and 25T-Keccak are having close hash performance in both GTX780 and GTX295. The main reason for this is that they share the same fine grain parallelism design that uses multiple threads to hash a copy of Keccak. The only difference is that 25T-Keccak requires more shared memory access to hash a copy of Keccak compare to 5T-Keccak. On the other hand, 5T-Keccak reuse more intermediate values to hash a plane or sheet of Keccak in one thread, hence it requires lesser shared memory access. When the tree height is greater, more tree leafs are hashed in parallel (L=D^H) [4], the shared memory traffic is also higher at this point. This explains why 5T-Keccak is slightly faster than 25T-Keccak when the tree height H>5, since it suffer lesser from the intensive shared memorry access.

Meanwhile, 1T-Keccak exhibit the fastest hash throughput compare to 5T-Keccak and 25T-Keccak. Since 1T-Keccak is hashing the entire Keccak within one thread, many intermediate state values and variables can be reused, which greatly reduced the memory read/write operations. In contrast, 5T-Keccak and 25T-Keccak need to use shared memory for sharing intermediate state values. Although shared memory is considered the second fastest memory in GPU after register, the additional read/write operations introduced by these two implementation techniques involved a lot of overhead; hence it slows down the overall performance. It is also noted that 1T-Keccak only outperform the other two implementations when the tree height reach certain level. This is due to the fact that when the tree height is low, 5T-Keccak and 25T-Keccak are able to launch more threads to hash concurrently, so the performance tends to be better compare to 1T-Keccak. When the tree height increases, the threads pool laucned also increases, memory access speed becomes the dominant factor that determine hash throughput, so 1T-Keccak that perform all computation locally will have the upper hand in this case.

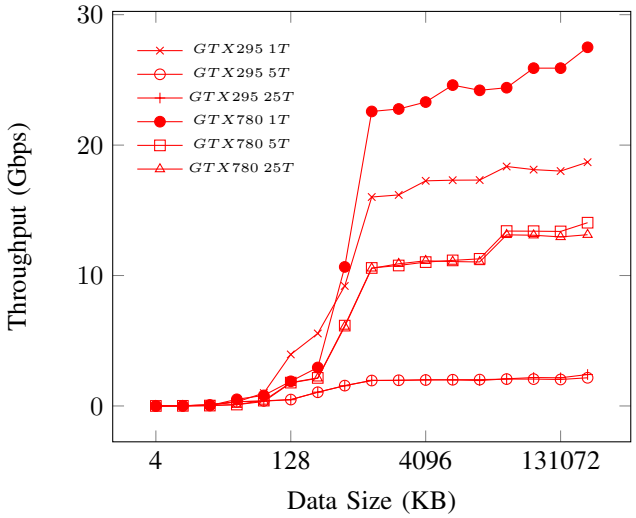


Fig. 11. Throughput in GTX295 and GTX780, with varying data sizes. The tree height was fixed at H=6. The work in GTX780 was implemented without Dynamic Parallelism.

Second experiment hash a single file with the various file size ranging from 4KB to 256MB. Figure 11 shows the effect of varying file size to the hash throughput. Since we are implementing LI tree-mode Keccak, the tree structure is fixed, so we need sufficiently large data size to fully load the tree strucuture. With our experimental setting of H=6 and D=4, there will be 4096 of leafs (L=D^H), each leaf hash at least a copy of Keccak (1024 bit). As a result, we can see the hash throughput is near to maximum when the file size is greater than 512KB. Further increasing the file size does not yield great performance improvement, as the tree structure is already fully loaded.

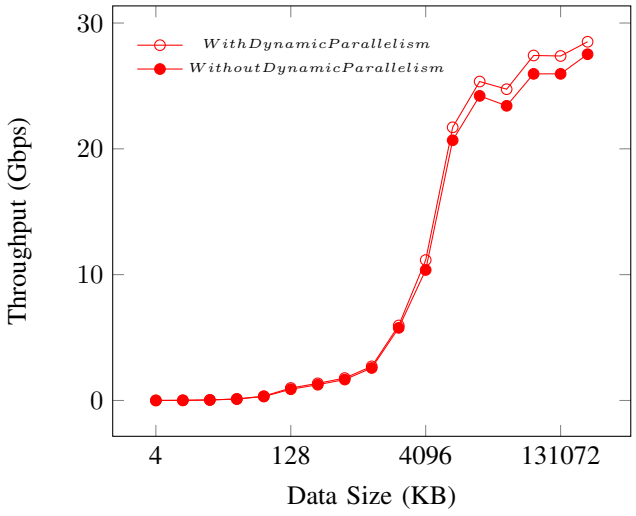


Fig. 12. Throughput in GTX780, implementation with and without Dynamic Parallelism

The results of using Dynamic Parallelism to manage the kernel launch for each tree levels is shown in Figure 12. By using Dynamic Parallelism, we are able to offload the kernel launch management to GPU itself, thus free up CPU to perform other tasks (e.g. hashing the top level). The maximum

TABLE II
COMPARISON OF OUR WORK WITH THE WORK IN [18], [19] AND [21]

Throughput (Gbps)					
H	Sevestre [19] (GTS250)	Cayrel *et al* [21] (GTX295)	Bos *et al* [18] (GTX295)	1T-Keccak (GTX780)	1T-Keccak (GTX295)
1	9.75 (Only implemented 2 levels of tree height)	0.08	17.70 (Tree height implemented is unknown)	0.38	0.19
2		0.57		1.80	0.86
3		1.29		5.73	1.01
4		1.89		7.04	1.91
5		N/A		19.34	7.22
6		N/A		28.51	18.69
7		N/A		27.63	17.96

throughput achieved with this technique is 28.51 Gbps. A slight improvement can be seen in the implementation using Dynamic Parallelism compare to the one without Dynamic Parallelism. However, we should be aware that the implementation using Dynamic Parallelism reduced the CPU workload in managing kernel launches, so it is useful in applications that demand more CPU computation. Hence, this technique is particularly useful for high traffic server environment where CPU may need to handle multiple tasks and heavy requests from clients.

From these experiments, we can conclude that in order to harness the GPU's parallel computing power, we need to provide sufficiently large data set for hashing. To achieve this, we can either hash a single large file in GPU, or group multiple smaller files into one large array in CPU before hashing it in GPU. For the latter case, we need to launch multiple tree structures to handle different small files with varying file size, which introduces additional overhead. Hence, the overall performance for hashing multiple small files may not be as good as hashing a large file. Another interesting implementation is to hash multiple files in batch mode [21], where each thread is assigned to hash a file. The actual implementation of this may need to consider the latency of launching multiple tree structures, queuing system of multiple files and the memory available in GPU. However, this is beyond the scope of this paper.

When comparing the hash throughput of our work with other researchers in Table III, our implementation is able to achieve 18.69 Gbps hash throughput in GTX295, which is 6% faster compare to the previously best known result by Bos *et al.* [18] that used the same platform. Our implementation in GTX780 utilizing Dynamic Parallelism is able to achieve 28.51 Gbps. On the other hand, Lowden *et al.* presented an optimized tree mode Keccak which is able to achieve 24 Gbps on a K20c GPU with Kepler architecture. Our 1T-Keccak implemented in GTX 780 is able to achieve 28.51 Gbps peak throughput, which is 18.8% faster than the implementation by Lowden *et al.* [22]. Moreover, both K20c and GTX 780 are from Kepler architecture, but K20c has 13 SMs (2496 cores) but GTX 780 only have 12 SMs (2304 cores); this implies that our proposed 1T-Keccak implementation can be even faster if implemented in K20c used by Lowden *et al.* [22].

VI. CASE STUDY: HIGH SPEED HASHING FOR TELEMEDICINE APPLICATIONS

In telemedicine applications, we often need to transmit large video data over Internet, either offline (as a file) or online (real time video conferencing). The size of medical video data can be very large (multiple GB range [40]); hashing such a large file is time consuming if it is performed using CPU. In such scenario, we can hash the large video data with GPU by using the techniques proposed earlier to achieve reasonable speed performance. For example, given the hash rate of GTX780 is 28.51 Gbps or 3.56 GBps (Table II), a video file of 2GB can be hashed within 0.56s only. Figure 13 shows that the large video file is first padded with '0' so that it's size is in multiple of 128 bytes; then the large padded file is divided into several smaller parts $P_0, P_1, ..., P_{n-1}$ where n is the number of leafs in a tree structure. The large video file can now be hashed in parallel using GPU.

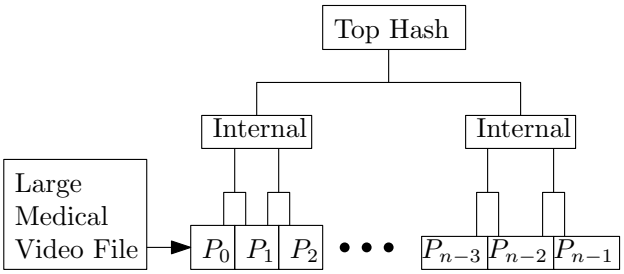


Fig. 13. Hashing a Large Video File

On the other hand, medical images are usually smaller in size (several MB to several hundreds MB). To hash these smaller images, we can group them into a single binary file, then hash this file in parallel using the tree-mode Keccak in GPU, just like the case in video file. Besides that, we can also hash multiple medical images independently, which is illustrated in Figure 14. The medical images are first padded to multiple of 128 bytes, then hashed in parallel by tree-mode Keccak.

To hash the telemedicine files (video, audio, image) in high speed, the sender first retrieve the data from database and transfer them to the GPU for parallel hashing using the techniques (1T-Keccak) described in Section IV. Once the hashing is completed, these files together with the top hash value, can be sent over to the receiver. The memory copy between CPU and GPU can be overlapped with parallel hashing in GPU (See Section V.F) to improve the overall performance. Upon receiving the files, the receiver can start computing the top hash value of all the received files and compare it against the received hash value. If any of the files was tampered or corrupted, the computed hash value is will be different with the received hash value and this can be detected immediately. Note that the files need to be hashed and transmitted in batches in order to fully utilize the massively parallel computational power in GPU. On top of that, Dynamic Parallelism can be useful in such application, as the hash tree kernel launch is now managed by GPU entirely (See Section V.B), leaving CPU free to execute other tasks.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2849439, IEEE Access

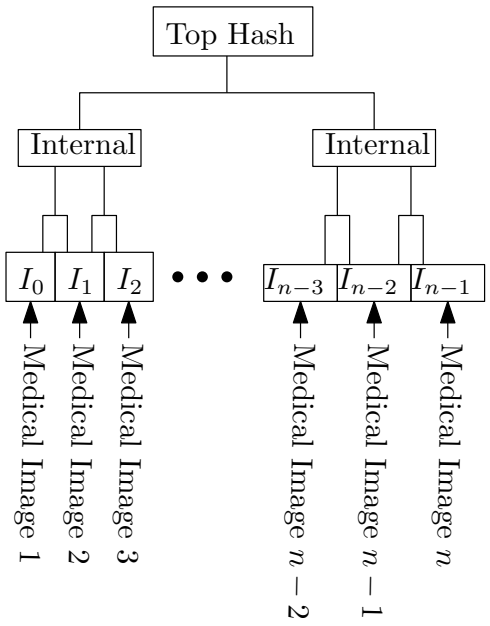


Fig. 14. Hashing Multiple Small Image Files

VII. Conclusion

In this work, we presented techniques to implement parallel and high speed hashing in GPU, which can be used to check the integrity of medical data transmitted over Internet for telemedicine applications. We first investigated the parallel granularity to implement LI tree-mode Keccak hash function in GPU, and demonstrated that one thread hashing one copy of Keccak is the best parallel granularity in GPU. Although other granularities (five threads per Keccak and 25 threads per Keccak) are able to exploit the inner parallelism of the Keccak hash function, they require the use of shared memory to share intermediate state and variables, hence increasing the memory read/write operations. In contrast, granularity of one thread per Keccak is able to reuse the intermediate state and variables during calculation, hence it is able to achieve faster hashing. By utilizing Dynamic Parallelism, the latest feature offered by NVIDIA GPU, we are able to offload the kernel launch management task to GPU, and free up CPU for other work. Although Dynamic Parallelism does not provide a very significant contribution to the overall performance, it does provide a framework to design applications that require high CPU computation in conjunction with GPU co-processing. We also proposed new optimization method to avoid bank conflicts when accessing shared memory. At the same time, data pre-fetch and loop optimizations (unroll and inversion) are used in our implementation, which can further improve the performance for GPU tree based implementation of Keccak. Our implementation result is also faster than all prior works from the literature.

The developed implementation techniques presented in this paper can also be used to protect other new form of networking topologies [39], including edge computing, fog computing and etc. Enhancing the hash rate under these new networking topologies (involving various hardware architectures) would be an interesting future direction we wish to pursue.

VIII. Acknowledgments

This work was supported by the Natural Science Foundation of China (No.61602118, No.61572010 and No.61472074), Fujian Normal University Innovative Research Team (No.IRTL1207), Natural Science Foundation of Fujian Province (No.2017J01738), the key project of the Sichuan Provincial Department of Education (No.17ZA0079).

References

[1] N. Xiong, X. Jia, L. T. Yang, A.V. Vasilakos, Y. Li, Y. Pan, A distributed efficient flow control scheme for multirate multicast networks, IEEE Transactions on Parallel and Distributed Systems, 21(9): 1254 1266, Feb 2010.

[2] N. Xiong, A. V. Vasilakosb, L. T. Yang, C. Wang, R. Kannane, C. Chang, Y. Pan, A novel self-tuning feedback controller for active queue management supporting TCP flows, Information Sciences, 180(11): 22492263, June 2010.

[3] M. Stevens, "Cryptanalysis of MD5 and SHA-1", *Special-Purpose Hardware for Attacking Cryptographic System*, SHARCS 2012.

[4] T. Xie, G. F. Deng and F. B. Liu, "A New Collision Differential For MD5 With Its Full Differential Path," in *Cryptology ePrint Archive*, Report 2008/230, 2008.

[5] X. Wang and H. Yu, "How to break MD5 and other hash functions", in *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of LNCS, pp. 561561. Springer Berlin/Heidelberg, 2005.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak, a SHA-3 candidate, 2009.

[7] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. BLAKE, a SHA-3 candidate, 2008.

[8] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas and J. Walker, The Skein Hash Function Family, a SHA-3 candidate, 2009.

[9] H. Wu. The Hash Function JH, a SHA-3 candidate, 2009.

[10] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlffer and S. S. Thomsen, Grøstl, a SHA-3 candidate, 2008.

[11] R. C. Merkle, "Secrecy, authentication, and public key systems", PhD thesis, Stanford University, 1979.

[12] P. Sarkar and P. J. Schellenberg, "A Parallelizable Design Principle for Cryptographic Hash Functions", in *Cryptology ePrint Archive*, Report 2002/031, 2002.

[13] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, Keccak Implementation Overview Version 3.2, 2012.

[14] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, Cryptographic Sponges. Available: http://sponge.noekeon.org/.

[15] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, The Keccak Reference Version 3.0, 2011.

[16] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak Sponge Function Family Main Document Version 2.1.

[17] CUDA C Programming Guide 8.0, 2017.

[18] J. W. Bos, D. Stefan, "Performance analysis of the SHA-3 candidates on exotic multi-core architectures", in *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2010.

[19] G. Sevestre, "Implementation of Keccak hash function in Tree hashing mode on Nvidia GPU", hpgu.org. Available: http://hgpu.org/?p=6833, 2010.

[20] G. Chindemi, N. Crovetti, Cuda-Keccak. Available: http://code.google.com/p/cuda-keccak/, 2011

[21] P. Cayrel, G. Hoffmann, M. Schneider, "GPU Implementation of the Keccak Hash Function Family", in *The 5th International Conference on Information Security and Assurance*, August 2011.

[22] J. Lowden, M. Lukowiak and S. Lopez Alarcon, "Design and performance analysis of efficient Keccak tree hashing on GPU architecture", Journal of Computer Security, vol. 23, no. 5, pp. 541-562, 2015.

[23] D. J. Bernstein, H. C. Chen, C. M. Cheng, T. Lange, R. Niederhagen, P. Schwabe, B. Y. Yang, "ECC2K-130 on NVIDIA GPUs", in *Progress in Cryptology - INDOCRYPT 2010*, pp. 328-346, 2010.

[24] S. Rennich, "CUDA C/C++ Streams and Concurrency". Available: http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf, 2011.

[25] B. Koziel, R. Azarderakhsh, M. Mozaffari-Kermani, "A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography", IEEE Transactions on Computers, in Press, 2018.
[26] W. Dai, D. D. Chen, C. C. Cheung, Cetin Kaya Koc, "FFT-based McLaughlin's Montgomery Exponentiation without Conditional Selections", IEEE Transactions on Computers, in Press, 2018.
[27] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, "Compact FPGA implementations of the five SHA-3 finalists", in *European Network of Excellence in Cryptology II (ECRYPT2) Hash Workshop*, 2011.
[28] A. Khajeh-Saeed, J. B. Perot, "Computational Fluid Dynamics Simulations Using Many Graphics Processors", *IEEE Journal of Computing in Science & Engineering, Volume 14 Issue 3, pp. 10-19*, May-June 2012.
[29] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A. Veidenbaum, "Efficient Simulation of Large-Scale Spiking Neural Networks Using CUDA Graphics Processors", *Proceeding of International Joint Conference in Neural Networks*, pp. 32013208, IEEE Press, 2009.
[30] P. Carpenter, "Accelerating Cryptographic Primitives with GPUs", hgpu.org. Available: http://www.auburn.edu/~carpept/security.pdf, Aug 2012.
[31] J. Won, S.-H. Seo, E. Bertino, "Certificateless Cryptographic Protocols for Efficient Drone-Based Smart City Applications", IEEE Access, vol. 5, pp. 3721-3749, 2017.
[32] J. W. Bos, D. A. Osvik, D. Stefan, "Fast Implementations of AES on Various Platforms", in *Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers (SPEED-CC)*, pp. 19-34. Berlin, Oct 2009.
[33] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine and G. Gogniat, "Hardware/Software Co-Design of an Accelerator for FV Homomorphic Encryption Scheme Using Karatsuba Algorithm", IEEE Transactions on Computers, vol. 67, no. 3, pp. 335-348, 2018.
[34] W. Pan, F. Zheng, Y. Zhao, W.-T. Zhu, J. Jing, "An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration", IEEE Transactions on Information Forensics and Security, vol. 12, no. 1, pp. 111 - 122, 2017.
[35] P. Martins, J. Eynard, J.-C. Bajard, L. Sousa, "Arithmetical Improvement of the Round-Off for Cryptosystems in High-Dimensional Lattices", IEEE Transactions on Computers, vol. 66, no. 12, pp. 2005-2018, 2017.
[36] W. Dai, Y. Dorz, Y. Polyakov, K. Rohloff, H. Sajjadpour, E. Sava, B. Sunar, "Implementation and Evaluation of a Lattice-Based Key-Policy ABE Scheme", IEEE Transactions on Information Forensics and Security, vol. 13, no. 5, pp. 1169-1184, 2018.
[37] X. Fei, K. Li, W. Yang, K. Li, "A secure and efficient file protecting system based on SHA3 and parallel AES", *Parallel Computing*, vol. 52, pp. 106-132, 2016.
[38] N. Xiong, A.V. Vasilakos, L.T. Yang, L. Song, Y. Pan, R. Kannan, Y. Li, "Comparative analysis of quality of service and memory usage for adaptive failure detectors in healthcare systems", IEEE Journal on Selected Areas in Communications, 27 (4), 495-509, 2009.
[39] Y. Zhou, D. Zhang, N. Xiong, "Post-Cloud Computing Paradigms: A Survey and Comparison", Tsinghua Science and Technology, vol. 22, no. 6, pp. 714-732, 2017.
[40] W. Xiang, G. Wang, M. Pickering, Y. Zhang, "Big video data for light-field-based 3D telemedicine", IEEE Network, vol. 30, no. 3, pp. 30-38, 2016.