Implementation of Keccak hash function in Tree hashing mode on Nvidia GPU

Guillaume Sevestre

guillaume.sevestre@gmail.com

Abstract. This paper presents a Graphics Processing Unit implementation of KECCAK cryptographic hash function, in a **parallel** tree hash mode to exploit the parallel compute capacity of the graphics cards. The Nvidia **Cuda language** has been used to access precisely the specificity of the GPU hardware (memory hierarchy, host-device memory transfers). After optimizations of the cooperation between GPU and CPU, top speed of more than **1 GB/s** (including data transfers) has been reach using an entry level GTS 250 card, for a **256 bits** security target *and* hash length. A Stream Cipher mode has also been implemented, which can find applications in high speed encryption or pseudo random number generation.

Keywords: Cryptographic hash function, KECCAK, GPU, Cuda, Tree hashing, Sponge functions, SHA-3 proposal.

1 Introduction

GPU computing applied to cryptography has already been explored, by porting for example the AES block cipher on this platform like in Manavski [5] and Osvik et al [4].

This paper presents a Graphics Processing Unit (GPU) implementation of the cryptographic hash function KECCAK [1], a candidate to the SHA-3 competition. To use efficiently the parallelism of the GPU architecture, a dedicated tree hashing mode is proposed for this implementation. This mode is inspired by different tree mode proposals in the KECCAK specification and in the MD6 hash function proposal [8].

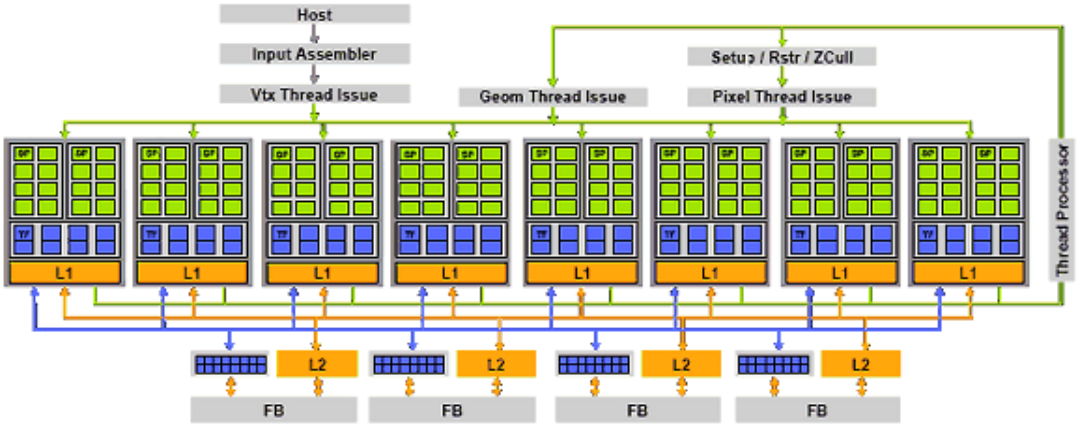
2 Targeted hardware and language framework

The targeted hardware for this implementation is Nvidia GPUs in general, and a G92 chip based card (GTS 250) is used by the author for benchmark. The choice of Nvidia Cuda API has been done to access precisely the specificity of the hardware (memory hierarchy, host-device memory transfers). The Cuda API is an extension to the C language, so programmers familiar with C can easily use this framework.

The parallel programming model used is Single-instruction multiple-thread (SIMT), which can be viewed as an extension of the Single-instruction multiple-data (SIMD) model (used by SSE and Altivec vector instructions set). In this model each thread process the *same* instructions on *different* data, by opposition to the standard Multiple-instruction multiple-data (MIMD) model used on multi-processor architecture, on which each thread can process *different* instructions on *different* data.

The G92 used 128 Cuda cores which are individually able to perform 32 bits integer operations, with 1024 32 bits registers available by core. Cores are grouped by 8 to form a Streaming MultiProcessor (SM), able to access some extra shared memory, 16 kB par SM. Each Streaming Multiprocessor is able to run near thousands of concurrent threads, and it is recommended to use at least hundreds threads by SM. Limitations comes by the size of the data processed by each thread, which should fit into available registers for better performance.

Fig. 1. Nvidia G92 hardware



Different memory areas are used by Cuda hardware, they are called device memory by opposition to host memory which resides on the host computer. The device global memory fits in VRAM, it is the largest memory available, but the slowest to access for cores. Shared memory is accessible by each core in a SM, and is faster than global memory. Shared memory can be used to store cooperation data used between threads during the computation.

Registers are private memory for each thread, and are very fast.

The typical workflow of a Cuda program is as follow:

1. Allocate memory workplace in host and device,
2. Transfer data from host to device global memory,
3. Load working memory in shared memory or registers,
4. Perform the core computation on GPU,
5. Transfer results back to the host memory,
6. Perform CPU finalization of computation (if needed).

Details on the programing model, the Cuda API, hardware configurations and capabilities, and performance recommendations can be found in the Cuda programming guide provided by Nvidia [7]. Details on how to install the Cuda SDK and toolkits can be found on the Cuda developer zone web site.

3 Choosing the hash mode

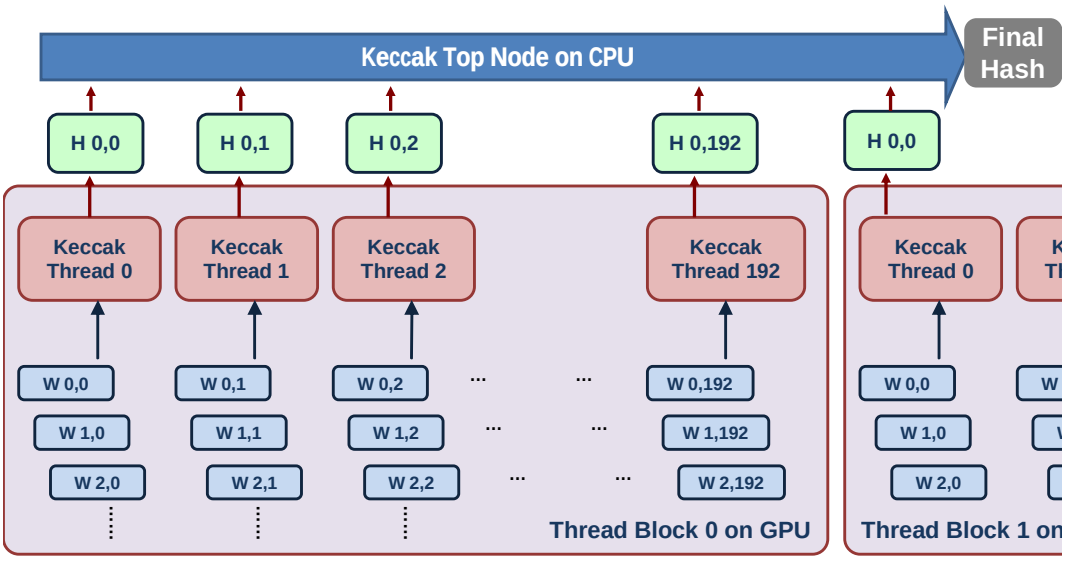
3.1 Tree hash mode proposed

Several hash functions proposed for the SHA-3 competition are designed to offer *inner* parallelism, in a sense that in the execution of one hash function call, different computation steps can be done in parallel. For example, in the MD6 design [8], 16 steps of the round function are independent and can be computed in parallel. One can try to exploit this inner parallelism on 'many threads' Cuda hardware.

An other way to parallelize *any* hash function is to use a tree hash mode, also called Merkle tree [6]. In this way the parallelism is done outside the hash function, by running several instances of the hash function concurrently, and gathering the results of each hash function by hashing them at an upper level in a tree.

The author chooses to implement a tree hash mode, with many leaves (reusing KECCAK paper [1], §6.4 vocabulary) hashing input message parts in the GPU, and a top node hashing results of leaves in a serial way on the Host CPU. A leaf interleaving mode have been chosen, but a fixed input message size for each leaf is used. So the GPU hashes fixed size input data 'big blocks' and return hash results to be processed by the *sequential* top hash node on CPU.

Fig. 2. Basic TreeHash mode



Each leaf is computed by a thread on the GPU, and threads are grouped in Threads-Blocks. Output of threads are transfers to CPU to the top hash node. This choice allows to define a streaming mode on big blocks of data, having the advantage to be stateless on the graphic card. On the other hand, the size of the big blocks (several MB) restrains this tree mode to big files only.

3.2 Memory ordering

To improve memory read/write speed, the Tree hash mode have been designed to ensure *coalesced* memory access, which means that, when threads are reading 32 bits words from global memory, in an array like $W[]$, $W[0]$ is read by Thread 0, $W[1]$ is read by Thread 1, $W[i]$ is read by Thread i, etc ... This memory ordering ensure best performance on old Nvidia Hardware (depends on compute capability of the GPU).

3.3 KECCAK version used

As targeted hardware device is limited in register memory, the author chooses to implement KECCAK-f[800] permutation. Using a compact implementation, a working hash state (hash state and extra working memory needed for computation) can fit into 41 32b registers, so as there are 1024 registers in a SM, up to 192 hash states can fit in a SM, and 192 threads can be launched in parallel.

To following parameters for the KECCAK sponge function have been chosen : KECCAK-f[800][r=256,c=544], and 256 bits of output size. The capacity of 544 bits and the output sizes enables to claim 256 bits security for the hash functions. The input rate of 256 bits, equal to output size, facilitates the tree construction as output data blocks are input data blocks of upper nodes.

It has to be noted that with bit rate divide by 4, and work factor divide by 2 (in approximation), KECCAK-f[800][r=256,c=544] is expected to be **2 times slower** than KECCAK-f[1600][r=1088,c=512] or KECCAK-f[1600][r=1024,c=576], which are respectively the KECCAK proposal for SHA-3 256 and the default KECCAK function proposed by KECCAK authors.

4 Performance results

4.1 First performance results

In this section first performance results are presented, using a basic tree mode as described before. In addition to the GTS 250 card, the author also use a laptop Cuda card (Quadro FX 370M), similar to the GTS 250 but with only 8 Cuda cores (16 times less than the desktop card). Comparing performance on this two cards allows to study scaling ability of the software. A full CPU implementation is also used for benchmark, using 32 bit mode on one core (no multithreading) and *without* SIMD (SSE) instructions. This implementation is also used to improve confidence in the correctness of the GPU implementation.

Table 1. First performance results

System	Core2 Duo 2.6 Ghz Quadro FX 370M	Core i5-750 2.6Ghz Nvidia GTS 250
CPU Hash speed in MB/s	**25**	**15**
CPU + GPU speed in MB/s	**61**	**682**

4.2 Improving performance

The first way to improve performance is to overlap GPU and CPU computation. As the kernels (function executed on the GPU) launches are asynchronous in the Cuda API, the CPU can compute the Keccak top node of the previous GPU computation results during the current GPU work.

In basic Cuda execution model, memory transfers (between host and device) and computation on GPU are done in sequence. Data transfers are slow and can be the bottleneck of the program performance. One other way to improve performance, if the hardware supports it, is to *overlap* data transfers and computation on GPU. This can be done using page-locked memory, and Cuda streams (succession of data transfers and computation that are issued in order in each stream), as explained in the Cuda programming guide [7]. The combination of overlapping GPU and CPU computation and overlapping data transfers and GPU computation gives the best results.

It's interesting to note that those improvements do not affect much the slower GPU configuration, as GPU computation time is still the most consuming task. In order to optimize performance using overlapping, the data transfers and works of the GPU should be divided in smaller independent (in terms of computation and data transfers) pieces, to be overlapped. But to fully occupied the several GPU cores those pieces should not be to small. Good trade off between smaller work size for overlapping and bigger work size to occupy all the GPU cores must be found.

Fig. 3. Overlapping data transfers, GPU and CPU computations

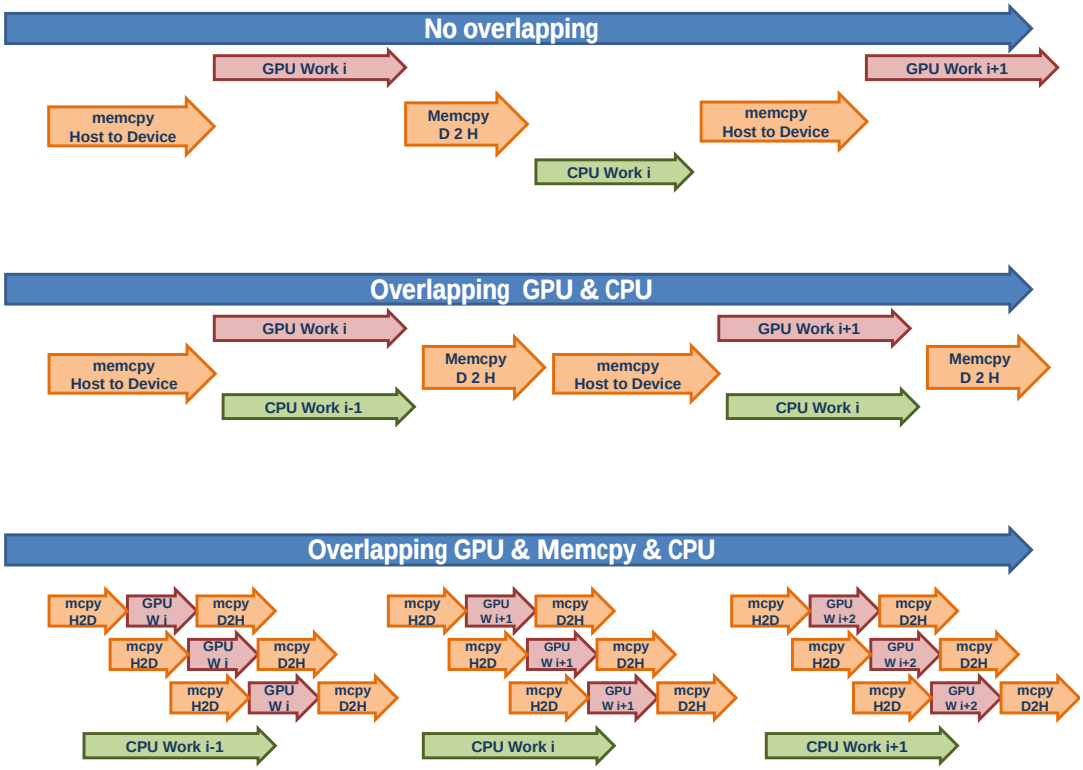


Table 2. Improved performance results

Systems Configuration	Core2 Duo 2.6 Ghz Quadro FX 370M	Core i5-750 2.6 Ghz Nvidia GTS 250
CPU Hash speed in MB/s	**25**	**15**
CPU + GPU speed in MB/s	**61**	**682**
CPU + GPU overlapped (MB/s)	**63**	**1032**
CPU + GPU Overlapped + Streams (MB/s)	**64**	**1219**

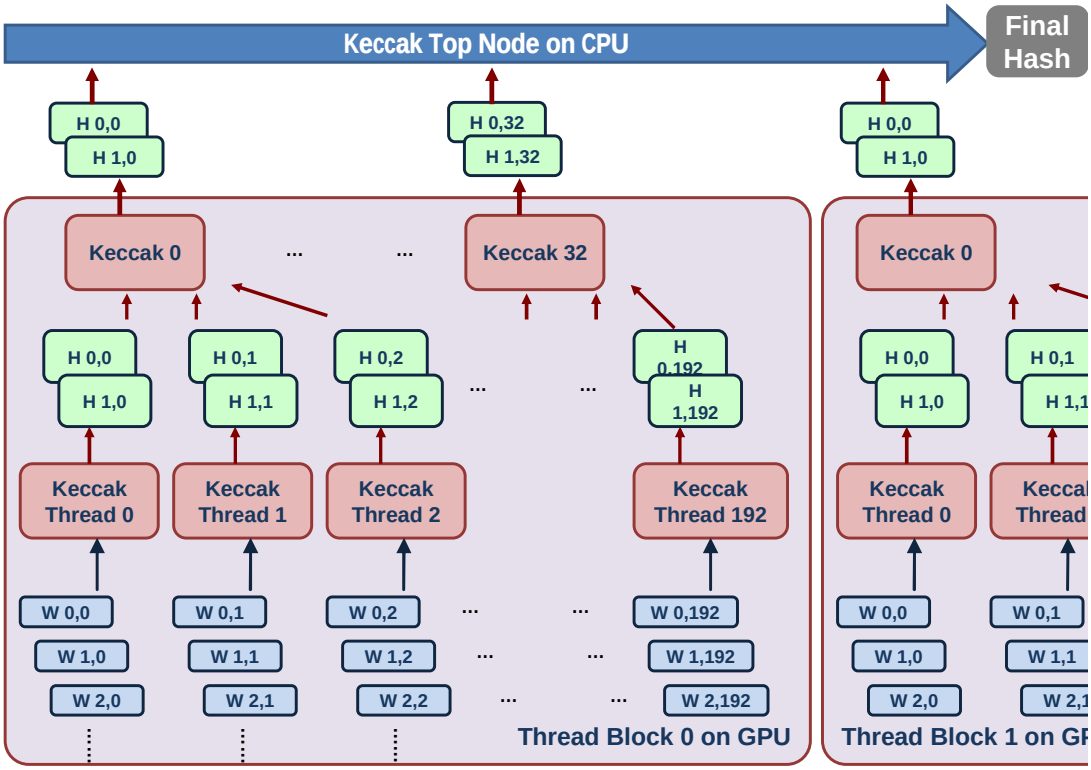
5 Enhanced Tree hash mode

5.1 Second Tree hash mode proposed

The first Tree hash mode proposed uses only 256 bits of chaining value between tree nodes. The aim of the second tree hash mode proposed is to use a double sized chaining value, as required in [2] as a necessary condition for being a *sound* tree hash mode. As doubling the chaining value size in first Tree hash mode proposed would double the load of the CPU top node, a tree of height 2 is used in the second mode.

Output of first nodes are store in shared memory, which enable the fewer threads computing the height 2 nodes to access to the heigth 1 nodes results. Shared memory visibility is limited to the thread block, so each thread block implement a subtree, and results of each subtrees are still hashed by the CPU top node.

Fig. 4. Second TreeHash mode



5.2 Performance of the second Tree hash mode

Performance of the second tree hash mode is slightly under the performance of the first mode proposed, but it's a security/performance trade off. All the asynchronous improvements of the first mode are also used in this mode.

Table 3. Performance results of the second Tree hash mode

Systems Configuration	Core2 Duo 2.6 Ghz Quadro FX 370M	Core i5-750 2.6 Ghz Nvidia GTS 250
CPU Hash speed in MB/s	**23**	**14**
CPU + GPU Overlapped + Streams (MB/s)	**59**	**1183**

5.3 Future work

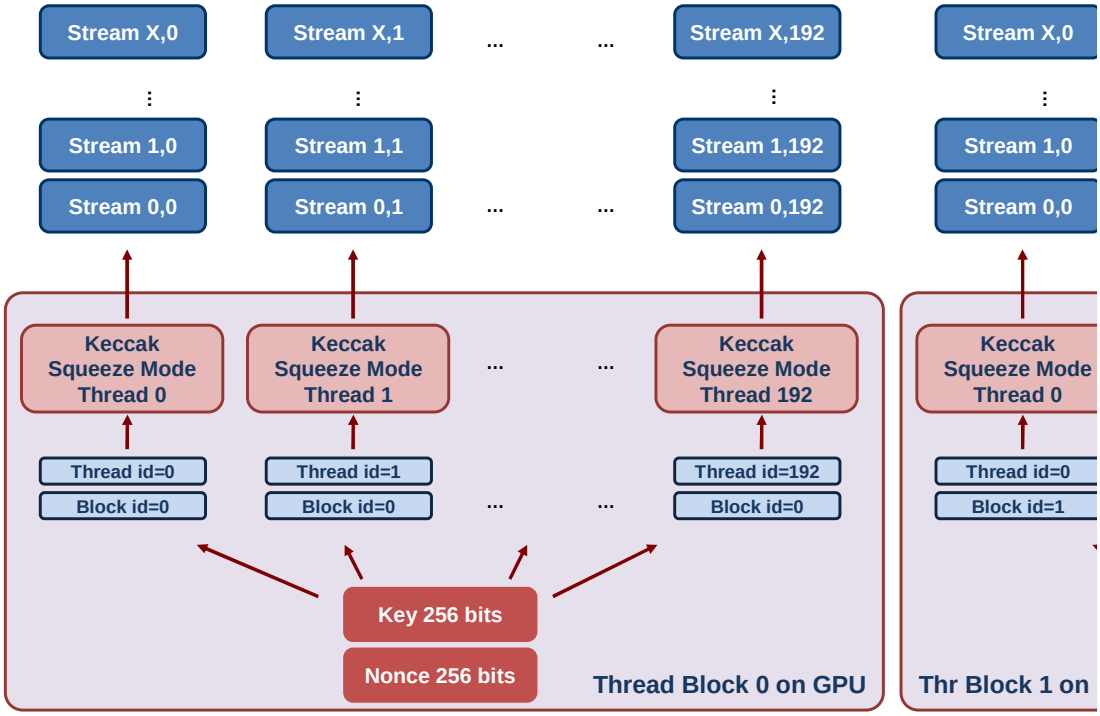
A proper streaming hashing API could be build over the functions implemented in this project. this API should handle padding of input message, and should be designed to fulfill the requirements of [2] to be a sound tree hash mode.

6 Stream Cipher and Pseudo Random Number Generator (PRNG)

6.1 Stream Cipher

The parallelism of the GPU can be fully used by implementing KECCAK in a Stream Cipher mode with independent streams generated by each threads in the GPU. In this mode, a 256 bits Key and a 256 bits (or less) Nonce is transfered from the Host to the Device. Then each thread hashes the Key and the Nonce, the thread id and the thread block number. Output streams are generated by using the arbitrary output length mode of KECCAK, also called *squeezing* mode. The whole computation is done on the GPU in this mode.

Fig. 5. Stream Cipher mode



6.2 Performance of Stream Cipher mode

Table 4. Performance of StreamCipher mode

Systems	Core2 Duo 2.6 Ghz	Core i5-750 2.6 Ghz
Configuration	Quadro FX 370M	Nvidia GTS 250
GPU using Cuda Streams (MB/s)	**62**	**1183**

The performance of stream cipher mode is quite similar to the first tree hash mode, since it involves the same number of operations to input X blocks of data in a KECCAK state in hashing mode than output X blocks of output in arbitrary output length mode.

The performance figures are not taking into account the XOR of the key stream with the clear text. It has to be tested if xoring clear text with key stream is more efficient on GPU (need to transfer clear text to GPU and cipher text back to Host) or on CPU.

6.3 Encryption and Authentication

If authentication is needed, a combination of StreamCipher mode and tree hashing for computing an HMAC over the data encrypted can be easily build. The sequence of operations could be

1. Transfer the Key and nonce to the GPU and start computing cipher key streams,
2. Asynchronously transfer clear text data to GPU,
3. XOR clear text with key streams in GPU,
4. Start computing HMAC in tree mode in GPU,
5. asynchronously transfer cipher text back to CPU,
6. when HMAC is computed, transfer result MAC to CPU.

This can be viewed as a parallel variant of the new *duplex sponge mode* proposed in [3].

6.4 High speed PRNG

The Stream Cipher mode can be used as a building block of an high speed PRNG. the Key and the Nonce can be replaced by true random data or data from a random *pool* to form a seed. The Stream Cipher function act as a random expander, returning 'cryptographic' random data from the seed.

7 Notes about overall performances and optimizations

7.1 Optimizations of KECCAK implementation and usage

Optimizations of KECCAK can be done on this project's implementation. For example lane complementing (described in [1]) §7.2) have not been used yet.

Overall performances figures in this paper can be greatly improved by using a better bitrate of KECCAK-f[800] , and/or use KECCAK-f[1600]. For KECCAK-f[1600], as GPU are 32 bits platforms, optimizations for using 32 bits words should be studied (bit interleaving).

7.2 Optimizations using new GPU hardware

Optimizations described in this paper, explicit asynchronous data transfers, coalesced memory access, are bound to the hardware used. New GPU hardware, in addition of generally having more Cuda cores embedded and more memory (registers, shared memory), can improve performance by doing implicit optimizations (for example: implicit asynchronous memory transfer between Host and GPU).

The author extrapolates that using new GTX 4XX Nvidia cards and KECCAK-f[1600], speeds of more than **3GB/s** should be within reach.

8 Disclaimer

This work is a *proof of concept* about using GPU for cryptographic hash, and it's still in alpha stage. Neither the *correctness* nor the *cryptographic strength* of this software is guaranteed.

9 Acknowledgments

The author thanks the KECCAK team for their comments and answers about this project.

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document. *Submission to NIST (updated)*, 2009.
2. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sufficient conditions for sound tree and sequential hashing modes. 2009.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. 2010.
4. J.W. Bos, D.A. Osvik, and D. Stefan. Fast Implementations of AES on Various Platforms. Technical report, Cryptology ePrint Archive, Report 2009/501, November 2009. http://eprint. iacr. org, 2009.
5. S.A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68. IEEE, 2008.
6. R. Merkle. A certified digital signature. In *Advances in CryptologyŮCRYPTOŠ89 Proceedings*, pages 218–238. Springer, 1990.
7. C. NVIDIA. programming guide version 3.0. *NVIDIA Corporation*, 2010.
8. R.L. Rivest, B. Agre, D.V. Bailey, C. Crutchfield, Y. Dodis, K.E. Fleming, A. Khan, J. Krishnamurthy, Y. Lin, L. Reyzin, et al. The MD6 hash function A proposal to NIST for SHA-3. *Submission to NIST*, 2008.