GPU Implementation of the Keccak Hash Function Family

Pierre-Louis Cayrel[1,2], Gerhard Hoffmann[2], and Michael Schneider[2]

[1] CASED – Center for Advanced Security Research Darmstadt, Germany
pierre-louis.cayrel@cased.de

[2] Technische Universität Darmstadt, Germany
hoffmann@mathematik.tu-darmstadt.de,
mischnei@cdc.informatik.tu-darmstadt.de

Abstract. Hash functions are one of the most important cryptographic primitives. Some of the currently employed hash functions like SHA-1 or MD5 are considered broken today. Therefore, in 2007 the US National Institute of Standards and Technology announced a competition for a new family of hash functions. Keccak is one of the five final candidates to be chosen as SHA-3 hash function standard. In this paper, we present an implementation of the Keccak hash function family on graphics cards, using NVIDIA's CUDA framework. Our implementation allows to choose one function out of the hash function family and hash arbitrary documents. In addition we present the first ready-to-use implementation of the *tree mode* of Keccak which is even more suitable for parallelization.

Keywords. Cryptography, Hash Functions, Keccak, GPU Computation

1 Introduction

Before the modern era, cryptography was concerned solely with message confidentiality, conversion of messages from a comprehensible form into an incomprehensible one and back again at the other end, rendering it unreadable by interceptors or eavesdroppers without secret knowledge (namely the key needed for decryption of that message). Encryption was used to (attempt to) ensure secrecy in communications, such as those of spies, military leaders, and diplomats. In recent decades, the field has expanded beyond confidentiality concerns to include techniques for message integrity checking, sender/receiver identity authentication, digital signatures, interactive proofs, and others.

Cryptographic hash functions are a specific type of cryptographic algorithm. Without the existence of hash functions, modern cryptography is not imaginary. Among many others, hash functions are required for digital signatures, message authentication codes, or password authentication. A hash function takes as input a bit string of arbitrary length, and outputs a short, fixed length hash value. Since existing hash functions like MD4, MD5, and SHA-1 are considered broken today, the NIST (US National Institute of Standards and Technology) announced in 2007 an open hash function competition. A new SHA-3 is supposed to replace

the older SHA-1 and SHA-2 hash functions. The competition started with 64 hash function candidates. 14 candidates were chosen for the second round. Now there have been five SHA-3 candidates selected by NIST for the third final round: BLAKE, Grøstl, Skein, JH, Keccak [7]. For a detailed classification of the SHA-3 candidates, see [11]. The main concern for selection into the final, third round was security. Nonetheless, performance has been a relevant factor as well. We chose Keccak for a GPU implementation, because it has been the fastest one out of the five remaining NIST SHA-3 candidates [10].

Brief Description of Keccak. This paper is about the Keccak family of hash functions, which is based on the sponge construction [6]. The construction consists of two phases: an *absorbing phase* and a *squeezing phase*. In the absorbing phase the variable-length input is padded and decomposed into blocks of equal length. Each block is then handled successively: first, it is XORed onto some fixed memory area (called the *state*), then an internal permutation is applied on the state. As long as there are input blocks left, this step is repeated. After that the construction switches into the squeezing phase, providing arbitrary-length output. To this end the permutation is repeatedly applied to the state. After each step, some bits of the state are read as additional output. The nominal version of Keccak operates on a 1600-bit state. There are 6 other state widths, though, ranging from 25 to 800.

Why a GPU Implementation. Since 2005 a transition to multiple-core CPU took place, it is no longer possible just to wait for a new CPU model to speed up a sequential application. Programs have to be designed with parallelism in mind to use the computational power of modern CPUs and modern graphic processing units (GPUs). Even more than the CPUs, modern graphic processing units have evolved into massively parallel processors. Their performance doubles every 12 to 18 months, exceeding even the rate of Moore's law [9, 15]. They are increasingly used for non-graphical calculations, an approach which is known as general-purpose GPU computing (GPGPU), or just GPU computing. As outlined in [9], in little more than five years a GPU will be equipped with more than 2.400 cores. Hardware companies are about to combine the computational power of the GPUs and the flexibility of multi-core CPUs into new architectures [12, 1]. Highly parallel machines will be quite common in the future.

Related Work. There are two other GPU implementations of Keccak available [17, 8]. Both only implement the tree-based mode proposed in [5], which will also be explained to some extent later in this document. The work of [17] is publicly available and implements Keccak-f[800]. Its implementation does not allow to hash arbitrary documents. The source code of [8] is closed at the moment, so we cannot compare to it. The Keccak version submitted to NIST is Keccak-f[1600]. Keccak-f[800] needs less memory and fits more naturally into the 32-bit GPU architecture. The prize to pay is that there is no publicly available reference implementation and that Keccak-f[800] possesses smaller values for the bit rate r and the capacity c. The bit rate r gives an estimation for the performance,

whereas the capacity c determines the security of Keccak. Details on these parameters will be given in Section 2.1. Another difference to our implementation is that there are as many copies of Keccak-f[800] running as there are threads. In our implementation 25 threads cooperate to execute only one copy of Keccak.

Our Contribution. As Keccak is the fastest of the five remaining SHA-3 candidates, it is natural to ask how well Keccak can be implemented on todays parallel machines. Specifically, we will show in a first step how the internal sponge permutation of Keccak can be parallelized on modern GPU models. Even a parallel version of Keccak's permutation can not overcome the fact that the sponge construction is an inherently sequential process, resulting in a very low occupancy of the GPU. We will address this restriction in a second step and show two approaches of how the GPU can be used more efficiently to implement Keccak, namely batch and tree modes. Furthermore, we provide a public implementation of the Keccak hash function family.

Organization of the Paper. The remainder of this paper is organized as follows. First, we will give a general overview of Keccak in Section 2. After introducing a pseudo-code version of Keccak's internal permutation, we will first give some details of the GPU in Section 3. Then we describe a GPU implementation of Keccak's internal permutation in Section 3. We discuss the batch-mode and the tree-mode, which allow a much higher usage of the GPU in Section 4. Finally, we present some experimental results in Section 5.

2 The Keccak Hash Function Family

Before introducing the Keccak hash function family, we give a short explanation of the security properties of hash functions.

2.1 Structure of Keccak

Keccak is a family of hash functions based on the sponge construction (cf. Figure 1), which is an iterated algorithm [6, 4]. The algorithm takes a variable-length

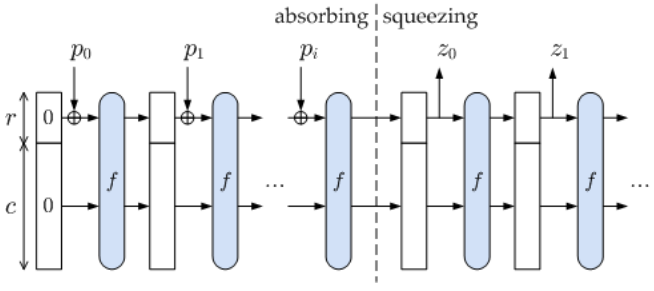

Fig. 1. The sponge construction [6]. p_i are the message blocks, f denotes the compression function, and z_i are the output blocks of the hash function.

bit string as input and produces an arbitrary-length output. The core of the algorithm is a permutation f repeatedly applied to a fixed-length *state* of $b = r + c$ bits. The value r is referred to as the *bit rate* and the value c as the *capacity*. Higher values of r improve the speed, higher values of c its security level. As described in [6], the sponge construction proceeds in the following steps:

(i) Reset the state to zero.
(ii) Pad the input message, such that its length is a multiple of r bits.
(iii) Absorbing phase: all input blocks of r bits are successively sent to the sponge construction in two steps: the r bits of each block (p_i) are XORed onto the first r bits of the state, which is then followed by an application of the permutation f.
(iv) Squeezing phase: After all blocks have been processed in the absorbing phase, the first r bits of the state are returned as output block (z_i). Should the user have requested more output blocks of r bits, then each repeated application of f will provide the next r-bit output block.

Note that the last c bits of the state are only changed by f itself, not by the XOR-operation in the absorbing phase. They are also never output in the squeezing phase.

2.2 Detailed View of Keccak

Seven possible Keccak hash functions form the Keccak family. They are denoted by Keccak-f[b], where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ is the width of the underlying permutation (in bits), which is also the width of the state A in the sponge construction. As noted above, $b = r + c$. The state consists of an array of 5×5 so-called *lanes*, each of length $w \in \{1, 2, 4, 8, 16, 32, 64\}$ bits, i.e. $b = 25w$. On a 64-bit processor, a lane of Keccak-f[1600] fits exactly in a 64-bit CPU word, which is why Keccak-f[1600] is the default version in the Keccak family. A more detailed pseudo-code of Keccak is shown in the full version.

2.3 Proposed Parameters

The recommended number of rounds for Keccak-f[1600] is 24. For SHA-3 the following parameter values have been proposed [2]: Keccak-f[$r = 1156, c = 448$], Keccak-f[$r = 1088, c = 512$], Keccak-f[$r = 832, c = 768$], Keccak-f[$r = 576, c = 1024$]. The default parameters are $r = 1024$ and $c = 576$, which are also the values we used in our implementation. The cost of finding a collision is 2^{288} binary operations. If an attacker has access to one billion computers, each performing one billion evaluations of Keccak-f per second, it would take about 1.6^{1061} years (1.1^{1051} times the estimated age of the universe) to evaluate the permutation 2^{288} times [3].

3 GPU Implementation

3.1 CUDA Programming Model

CUDA (*Compute Unified Device Architecture*) [14] is a GPGPU technology (*General-Purpose Computing on Graphics Processing Units*). Instead of executing an application exclusively on the central processor (CPU), some computational intensive parts of the application can be transferred to the graphic processor (GPU). Using the GPU for high-performance computing has been in practice for years already, but the lack of a suitable API made it a painstaking experience for the programmer, formulating his ideas in an API designed for pure graphics programming. In contrast, CUDA uses as API an extension of C or Fortran, which makes general-purpose programming on the GPU a lot easier. However, to program the GPU efficiently, a good knowledge of the internal workings of the GPU is still necessary.

Some NVIDIA GPU Internals. An extensive description of hardware details of NVIDIA GPUs can be found at [14, 15], and we will restate some of the main points here in the context of the GPU model used for this paper, the GTX 295. But even for the new GPU architecture, called Fermi, the description will be adequate. At the highest level, the GPU consists of a number of so-called streaming multiprocessors (SM). Each SM contains a fixed number of so-called streaming processors (SP), typically 8. The GPU supports hardware-multithreading: a stalled thread waiting for data can be quickly replaced by another thread in exactly one processor (SP) cycle. The threads are organized in bigger units, so-called *warps*, which typically consist of 32 threads. Warps are the units which are actually scheduled on an SM. They are in turn organized in blocks (*cooperative thread arrays (CTAs)*) of typical sizes of 128 up to 256 threads or 4 to 8 warps. Threads belonging to the same CTA can communicate via a special and very fast memory area called *shared memory*. There is no way for the CTAs themselves to communicate: CUDA requires the thread blocks to be independent to allow them to be executed in any order, which leads to portability and scalability of CUDA programs. Using shared memory is in theory as fast as using registers. In practice, some care is advisable, though. Shared memory is composed of DRAM: each read or write operation has to be followed by a refresh cycle. To allow for maximal parallelism, shared memory is therefore built up of so-called *memory banks*. As long as different threads of a warp (more exactly, of a half-warp) access different memory banks, there is no problem. Otherwise so-called *bank conflicts* occur, leading to a contention situation, which reduces performance. Another important fact for the GPU implementation is that threads in the same warp are always synchronized. That means that data written to shared memory by the threads of a warp are visible to all threads in the warp in the next computational step. Finally, the code to be executed on the GPU is called a *kernel*. It is a C (or Fortran) function which has to be called on the CPU. With older GPU models only one kernel can be active on the GPU at the same moment.

3.2 Basic GPU Implementation of Keccak on the GPU

It is inherently difficult to parallelize Keccak on the GPU. The main reason is the very nature of the sponge construction. For instance, if using Keccak-f[1600] with bit rate $r = 1024$ and capacity $c = 576$, then each new 1024-bit part of the data to be hashed is XORed onto the first 1024 bits of the current state A, leading to a very sequential process. Before addressing this limitation, we will show how to parallelize the main functionality of Keccak, namely the rounds of the hash permutation. We give here C-code for a CUDA kernel, which executes a 64-bit version of Keccak-f[1600] with only one warp consisting of 25 threads.[3]. Although the GTX 295 is a 32-bit machine, the 64-bit version performed better. The 32-bit version uses some extra tables and operations to reduce the number of bank conflicts, but also extends its source code.

Source Listing (Algorithm 1). GPU code is typically very susceptible against conditionals for this often results in divergence of threads. Also, all the calculations in Keccak are done modulo 5, which is an expensive operation on the GPU. Therefore we designed some tables to get a compact code with implicit modulo settings, which will then be executed by 25 threads in parallel. The tables are given in the full version of this paper and have to be copied from the host to the GPU at runtime. Once the tables are in place, the actual code for Keccak-f[1600] consists of only 5 lines of code. As usual, A denotes the state of the permutation, consisting of $1600/8 = 200$ bytes. C and D are temporary variables. Only the innermost permutation step of Keccak-f[1600] consisting of 24 rounds is shown. Each round itself consists of five steps named θ, ρ, π, χ, and ι, respectively. When loading more than $1600/8 = 200$ byte, or when switching to the squeezing phase of Keccak, it has to be called repeatedly.

Lines 9 and 10 of Algorithm 1 represent the first part of the θ step. Using the index s into A results in the same access pattern as using $A[\text{index}(x, y)]$, where x and y run from 0 to 4 and $index(x, y)$ as given in Line 3. As the line is executed by 25 threads accessing only 5 different memory locations, bank conflicts occur. They could be resolved by the use of another table, but runtime measurements have shown that performance does not improve substantially. Bank conflicts in Lines $10-11$ are more serious, but also more difficult to avoid due to the general access pattern of Keccak. For instance, the tables are kept in constant device memory. While this type of memory is cached, it is also afflicted with bank conflicts. Line 11 is made up of the last part of the θ operation. Using to the encoding tables, it also contains ρ and π. Finally, Line 12 represents χ, whereas Line 13 is the ι operation.

4 Additional Keccak Modes on the GPU

Hashing a single document with Keccak is an inherently sequential process. The GPU supports thousands of threads in hardware, so the question arises how to

[3] Even when started with 25 threads, the runtime system of CUDA will always make sure that a warp has 32 threads.

Algorithm 1 Keccak-f[1600], 64-bit, r=1024, c=576

INPUT: *data*, A
OUTPUT: The scrambled state A.

```
#define ROUNDS (24)
#define R64(a,b,c) (((a) ≪ b) ^ ((a) ≫ c))
#define index(x,y) (((x)%5)+5*((y)%5))
int const t = threadIdx.x;      int const s = threadIdx.x%5;
__shared__ uint64_t A[25], C[25], D[25];
if t < 25 then
    A[t] ← data[t];
    for i ← 0 to ROUNDS do
        C[t] ← A[s] ^ A[s + 5] ^ A[s + 10] ^ A[s + 15] ^ A[s + 20];   ▷ First part of θ.
        D[t] ← C[b[20 + s]] ^ R64(C[b[5 + s]], 1, 63);        ▷ End of the θ operation.
        C[t] ← R64(A[a[t]] ^ D[b[t]], ro[t][0], ro[t][1]);
        A[d[t]] ← C[c[t][0]] ^ ((∼ C[c[t][1]]) & C[c[t][2]]);                ▷ χ operation.
        A[t] ← A[t] ^ rc[(t == 0) ? 0 : 1][i];                               ▷ ι operation.
    end for
end if
```

use more than just the 25 threads as shown above. One possibility is to hash more than one document at once in a *batch mode*. This way, the above kernel is applied on many data streams on the GPU, thereby greatly enhancing the achieved parallelism. Another approach is hashing a single document in *tree mode*. In this case, parts of the document are first loaded into the leaves of a tree. These leaves are then hashed using the basic kernel above. The result values are pushed onto the next level in the tree, iterating the process.

Batch Mode. For the batch mode, create a CUDA kernel consisting of 128 threads, say. Each of the 8 warps executes the basic Keccak kernel above on a different document. Thus a high occupancy of the GPU is possible. Some care is necessary, though. One point to keep in mind is that the GPU has considerable less memory than the CPU. As thousands of documents might not fit in GPU memory, some kind of ordering is necessary on the GPU side. Such an ordering can be achieved using CUDA streams. Together with the ability of the new Fermi GPUs to start kernels concurrently, such a batch mode seems to be a viable option. Due to time reasons we did not include the batch mode to our implementation.

Tree Mode. The task of parallelizing cryptographic hash functions by using trees has been studied by several authors [16, 13]. The idea is to distribute the input data into the leaves of a tree and to hash those leaves independently and in parallel. Their hash values are combined and sent to corresponding parent nodes. The process is then applied recursively, until the root node provides the final digest. The Keccak main document [5] describes two possible approaches for such a tree-based mode: LI (*leaf interleaving*) and FNG (*final node growing*). We implemented only LI, for more information on FNG see the main document [5]. With LI, the tree size and the number of its leaves is a fixed number. The

tree itself is a balanced tree of height H. All internal nodes have the same degree D. The number of the leaves is then $L = D^H$. As the tree is fixed, its parameters can be provided as input parameters. For possible parameter ranges we refer to [5]. The data input are two binary strings: a prefix P, which can be used as a key or a salt, and the document to be hashed [5, Page 30]. In a first step, the prefix and the document are concatenated and padded to form the input message M, whose blocks are then interleaved onto the leaves, c.f. Algorithm 2.

Algorithm 2 Leaf-Interleaving

INPUT: Bit string M, the input message. A tree with $L = D^H$ leaves. B denotes the leaf block size (in bits).
OUTPUT: A tree with M interleaved into its leaves.
For each leaf L_j, $0 \leq j \leq L-1$, set L_j to the empty string.
for $i \leftarrow 0$ **to** $|M| - 1$ **do**
$j \leftarrow floor(i/B) \mod L$
Append bit i of M to L_j
end for

Note that it does not say that a leaf can only load B bits of data. Instead, the number B is used merely as a parameter for the interleave technique. Once the leaves are initialized, each one is hashed independently in parallel. As in our setting $D = 4$, the hash values of 4 leaves are concatenated to form the input for the associated parent node. The process is then continued recursively up to the root node, which provides the final hash value. By applying Keccak on the root node repeatedly, arbitrary-length output is possible.

5 Experimental Results

We made some tests on an NVIDIA GTX 295 GPU. The source code of our implementation is publicly available[4]. Timings for different tree parameters and document file sizes are given in Table 1 as reported by the CUDA profiler. The numbers are given in seconds. For comparison, we ran the standard hash function MD5 on the same files on an Intel(R) Core(TM)2 Duo CPU E8400 (3.00GHz) running Linux 2.6.32. We cannot compare to other GPU versions of Keccak, since they are either not public or do not allow to hash arbitrary documents. Table 1 shows the strength of the tree mode. For height H=0 (the original serial mode) the timings cannot compete with MD5, whereas Keccak-f[1600] with height H=4 is nearly as fast as MD5. Recall that MD5 is considered outdated and insecure, whereas Keccak is supposed to guarantee strong security.

Together with the SHA-3 submission, a reference implementation of Keccak was published. Table 2 presents timings we gained using the Keccak reference implementation in version 2.3. The experiments were performed on an Intel(R) Core2 Duo CPU E8400 running at 3.00GHz. Hashing 128 MB costs approximately half a second, for example. Tests on a Tesla C2050 GPU showed comparable results between MD5 and the GPU implementation of Keccak.

[4] `http://www.cayrel.net/spip.php?article189`

File size [bytes]	H = 0	H = 1	H = 2	H = 3	H = 4	MD5
1.050.112	0.415	0.104	0.020	0.014	0.019	0.003
10.500.096	4.110	0.994	0.144	0.069	0.063	0.025
25.200.000	9.854	2.375	0.332	0.151	0.129	0.057
50.400.000	19.702	4.742	0.655	0.291	0.199	0.112

Table 1. Timings in seconds for Keccak-f[1600] in tree mode on a GTX 295, for different tree heights H.

File size [bytes]	Cycles	Time [ms]
128.000	1.664.892	0.555
1.280.000	16.616.619	5.539
12.800.000	166.548.357	55.516
128.000.000	1.668.329.037	556.110

Table 2. Timings for the Keccak reference implementation on CPU, using a bit rate of r = 1024. We hashed blocks of size 128 byte each.

6 Conclusion and Further Work

The crucial function of Keccak is its internal permutation f, therefore an efficient implementation is very important. Its implementation (Algorithm 1) has two weaknesses: it only uses 25 threads of the 32 available ones for a warp, and it produces a relatively high number of bank conflicts. The basic scheduling unit of CUDA is the warp. Therefore, using the other seven threads in the execution of another permutation instance raises synchronization problems. As threads in the same warp are always synchronized, Algorithm 1 does not suffer from this issue. The bank conflicts themselves can be reduced using a 32-bit version of Algorithm 1, but is still not at all conflict-free. A deeper analysis of Keccak could reveal the path to a conflict-free permutation kernel. According to the CUDA profiler, Algorithm 1 executes in about $58\,\mu s$ (24 rounds, bit rate $r = 1024$), producing more than 9.000 bank conflicts. A corresponding 32-bit version of Algorithm 1 executes in about $50\mu s$ with a number of bank conflicts of ~ 1250. Another issue is that at the moment we have not used so-called *streams* and *Zero Copy*. With streams it is possible to copy data to the device or to the host asynchronously to kernel execution, whereas with Zero Copy the GPU can directly access host memory, thereby copying data to the device on demand. Zero Copy is especially interesting for integrated devices, e.g. in laptops. NVIDIA's new Fermi architecture could also be helpful here, for those new cards have more shared memory and a dedicated L1 cache, which might not suffer from bank conflicts. Fermi cards have 48 KB shared memory and 16 KB of dedicated L1 cache per SM, whereas the GTX 295 only has 16 KB of shared memory. Fermi cards also allow for more threads per SM. As the GPU scales nearly optimal, we expect to observe corresponding speedups in Keccak's runtime on the GPU.

References

1. AMD. AMD fusion family of APUs. http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf, 2010.
2. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Note on keccak parameters and usage. http://keccak.noekeon.org/NoteOnKeccakParametersAndUsage.pdf.
3. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Tune keccak to your requirements. http://keccak.noekeon.org/tune.html.
4. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. ECRYPT hash workshop, 2007.
5. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak sponge function family - main document. In submission to NIST (Round 2), 2009. http://keccak.noekeon.org/Keccak-main-2.1.pdf.
6. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponges. http://sponge.noekeon.org/, 2011.
7. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak sponge function family. http://keccak.noekeon.org/, 2011.
8. Joppe W. Bos and Deian Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In *CHES*, volume 6225 of *LNCS*, pages 279–293. Springer, 2010.
9. B. Dally. The future of GPU computing. http://www.nvidia.com/content/GTC/documents/SC09_Dally.pdf, 2009.
10. ECRYPT. SHA-3 hardware implementations. http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations, 2011.
11. Ewan Fleischmann, Christian Forler, and Michael Gorski. Classification of the SHA-3 candidates. http://eprint.iacr.org/2008/511, 2008.
12. Intel. Sandy bridge. http://software.intel.com/en-us/articles/sandy-bridge/, 2011.
13. R. C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford University, 1979.
14. NVIDIA. NVIDIA CUDA programming guide 3.2. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, 2010.
15. David A. Patterson and John L. Hennessey. *Computer Organization And Design*. Morgan Kaufmann, Burlington, USA, 2009.
16. Palash Sarkar and Paul J. Schellenberg. A parallelizable design principle for cryptographic hash functions. Cryptology ePrint Archive, Report 2002/031, 2002. http://eprint.iacr.org/.
17. Guilaume Sevestre. Keccak tree hashing on GPU, using NVIDIA CUDA API. http://sites.google.com/site/keccaktreegpu/, 2010.