Exploiting Heterogeneous Systems: Keccak on OpenCL

Allan Mariano de Souza [1], Fábio Dacêncio Pereira [1], and Edward David Moreno [2]
[1] Department of Computer Science/ COMPSI,University Center Euripides of Marília, Marília, Brazil
[2]Department of Computer Science/DCOMP, Federal University of Sergipe, Aracaju, Brazil

Abstract - *Using graphics processing units (GPUs) in high-performance parallel computing continues to become more prevalent, often as part of a heterogeneous system. CUDA and OpenCL are APIs and enables programmers to developer GPGPU applications and softwares to massively parallel processors. In October 2, 2012, NIST announced the winner of its five-year competition to select a new cryptographic hash algorithm, one of the fundamental tools of modern information security. This work is proposed to explore the winner algorithm of the SHA-3 competition, the Keccak, and subsequently implement the propose heterogeneous platform architecture on OpenCL with intuit to obtain performance data. Finally, will be compared OpenCL implementation of keccak with CPU and GPU execution.*

Keywords: GPGPU; OpenCL; Heterogeneous Systems; SHA-3 Keccak;

1 Introduction

In recent years, more and more multi-core/many-core processors are superseding sequential ones. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth, and this trends likely to continue [1]. Particularly, today's GPUs (Graphic Processing Units), greatly outperforming CPUs in arithmetic throughput and memory bandwidth, can use hundreds of parallel processor cores to execute tens of thousands of parallel threads [2]. Researchers and developers are becoming increasingly interested in harnessing this power for general purpose computing, an effort known collectively as GPGPU (General-Purpose computing on the GPU)[3], to rapidly solve large problems with substantial inherent parallelism.

CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) are APIs and enables programmers to developer GPGPU applications and softwares to massively parallel processors.

One of the methods to ensure information integrity is the use of hash functions, which generates a stream of bytes (hash) which must be unique. But most functions can no longer prevent malicious attacks and ensure that the information have just a hash. In order to solve this problem, the National Institute of Standards and Technology (NIST) convened the scientific community through a competition to create a new hash function standard, called SHA-3.

NIST received significant feedback from the cryptographic community. Based on the public feedback and internal reviews of the second-round candidates, NIST selected five SHA-3 finalists - BLAKE, Grøstl, JH, Keccak, and Skein to advance to the final round of the competition on December 9, 2010, which ended the second round of the competition[6].

In October 2, 2012, NIST announced the winner of the SHA-3 competition and the winner was Keccak and now will become official NIST's SHA-3 hash algorithm.

In this context, this work aims to study the winner SHA-3 algorithm, The keccak and then propose an implementation for heterogeneous systems using OpenCL to obtain performance data and comparison with CPU and GPU execution.

2 CUDA vs OpenCL

CUDA and OpenCL are fast, and on GPU devices they are much faster than the CPU for data-parallel codes, with 10X speedups commonly seen on data-parallel problems. Both CUDA and OpenCL can fully utilize the hardware. They are both entirely sufficient to extract all the performance available in whatever hardware device

Both CUDA and OpenCL can fully utilize the hardware. They are both entirely sufficient to extract all the performance available in whatever hardware device. Both OpenCL and CUDA provide a general-purpose model for data parallelism as well as low-level access to hardware, but only OpenCL provides an open, industry-standard framework. As such, it has garnered support from nearly all processor manufacturers including AMD, Intel, and NVIDIA, as well as others that serve the mobile and embedded computing markets. As a result, applications developed in OpenCL are now portable across a variety of GPUs and CPUs.

Spafford's ran ORNL's Scalable Heterogeneous Computing Benchmark Suite (SHOC) that has been optimized for both CUDA and OpenCL, and found that OpenCL can match CUDA performance on most of the basic math kernels[15].

GPU software maker AccelerEyes has seen CUDA and OpenCL performance equalize. The company, which recently released OpenCL-powered beta versions of their two flagship software products, ArrayFire and Jacket, has found

that for most kernel codes, the two technologies now exhibit similar performance[15].

The Future Technology Group at Oak Ridge National Lab (ORNL), has been benchmarking the two technologies for some time and is now convinced that OpenCL performance is now on par with that of CUDA. The figure 2.1 shows the results of the benchmarking.

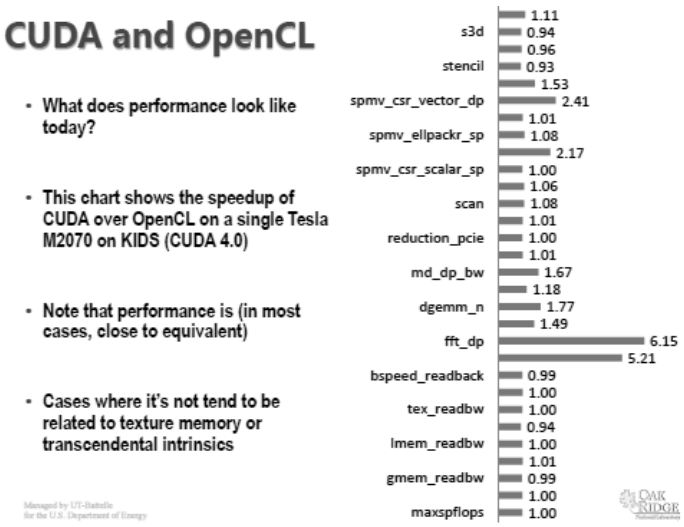

Figure 2.1: Benchmarking of performance CUDA and OpenCL [15]

Due to the high portability across a variety of GPUs and CPUs, the high performance power and your growing of OpenCL. This paper present an proposed implementation of keccak's algorithm for a heterogeneous systems using OpenCL.

3 OpenCL

OpenCL is an industry standard cross-platform and parallel-computing for programming heterogeneous applications that can be formed collection of CPUs, GPUs and other computing devices organized into a single platform. It's more than a language, OpenCL is an framework for parallel programming and includes a language, API, libraries an runtime system to support software development [4].

Single programs written on OpenCL can run on a wide range of systems, from cell phones, to laptops, to nodes in massive super-computers. No other parallel programming standard has such a wide reach [5].

The core idea behind OpenCL can be describe using follow hierarchy models. Platform model(3.1), execution model(3.2), memory model(3.3) and programming model(3.4).

3.1 Platform Model

The platform model consists of a host that are connected to one or more OpenCL devices (CPUs, GPUs, PDAs), The OpenCL devices are divided into one ore more compute units (CUs) which are further divided into one or more processing elements (PEs). The computations that are executed on OpenCL devices occur within the processing elements [4].

The figure 3.1 illustrate the OpenCL platform model that was described.

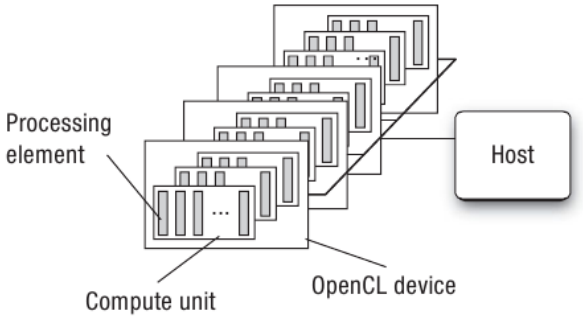

Figure 3.1: OpenCL Platform Model [5].

3.2 Platform Model

Execution of an OpenCL program occurs in two parts: kernels that are parallel parts or functions executed on one or more OpenCL devices and a host program serial parts executed on the host. The host program defines the context and parameters for kernels and manages their execution [4].

The core of the OpenCL execution is defined by how kernels are executed. When the host program submits a kernel for execution an index space are defined called NDRange, where these index can be one dimensional (1D), tow dimensional (2D) or three dimensional (3D). Each point in these index space are called work-item and each work-item are an instance of the kernel and each work-item has index (global ID) to compute memory addresses and make control decisions.

Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit [4].

The figure 3.2 are an example of how the global IDs, local IDs, and work-groups indices are related for a two-dimensional NDRange. Other parameters of the index space are defined in the figure. The shaded block has a global ID of (gx, g y) = (6, 5) and a work-group plus local ID of (wx, w y) = (1, 1) and (lx, ly) =(2, 1).

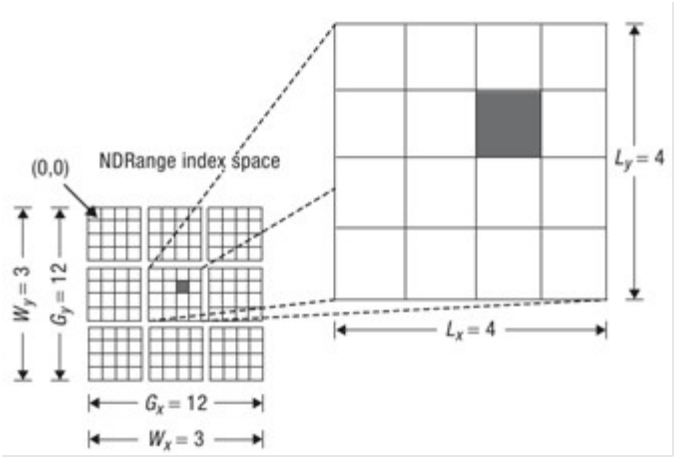

Figure 3.2: OpenCL Execution Model [5].

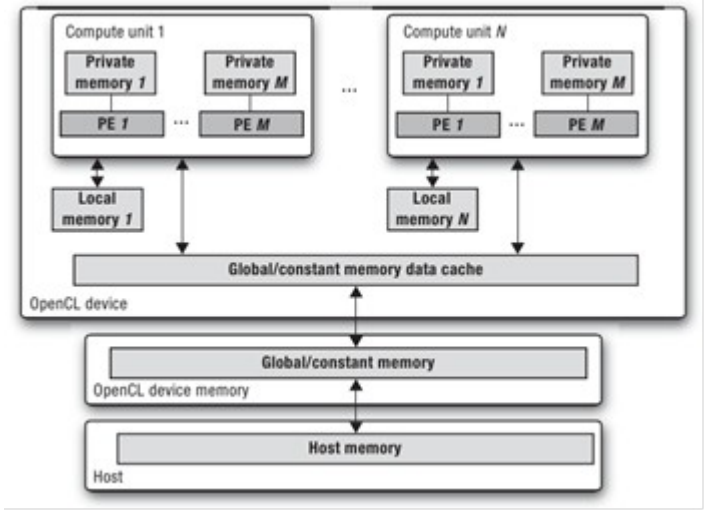

Figure 3.3: OpenCL Memory Model [5].

3.3 Memory Model

Work-items executing a kernel have access a five distinct memory regions [5].

- **Host memory:** This memory region is visible only to the host. As with most details concerning the host, OpenCL defines only how the host memory interacts with OpenCL objects and constructs.
- **Global Memory:** This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.
- **Constant memory:** This memory region of global memory remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory. Work-items have read-only access to these objects.
- **Local memory:** This memory region is local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.
- **Private memory:** This region of memory is private to a work-item. Variables defined in one work-item's private memory are not visible to other work-items.

The figure 3.3 shows a summary of the memory model in OpenCL and how the different memory regions interact with the platform model.

3.4 Programming Model

OpenCL includes an language based on C99 to write the kernel code, and the host program can be written in some other languages such as: C/C++, Java and Python. The OpenCL programming model supports data parallel and task parallel programming models, as well as supporting hybrids of these tow models.

4 Keccak Algorithm

The design philosophy of Keccak is the hermetic sponge strategy [7]. It uses the sponge construction for having provable security against all generic attacks. It calls a permutation that should not have structural properties with the exception of a compact description[8].

Keccak is a family of hash functions that is based on the sponge construction, and hence is a sponge function family. In Keccak, the underlying function is a permutation chosen in a set of seven Keccak-f permutations, denoted Keccak-f[b], where b ∈ {25, 50, 100, 200, 400, 800, 1600} is the width of the permutation. The width of the permutation is also the width of the state in the sponge construction[9].

The state is organized as an array of 5×5 lanes, each of length w ∈ {1, 2, 4, 8, 16, 32, 64} (b=25w). When implemented on a 64-bit processor, a lane of Keccak-f[1600] can be represented as a 64-bit CPU word. For obtain the Keccak[r,c] sponge function, with parameters capacity c and bitrate r, if we apply the sponge construction to Keccak-f[r+c] and by applying a specific padding to the message input.

All the operations on the indices are done modulo 5. A denotes the complete permutation state array, and A[x,y] denotes a particular lane in that state. B[x,y], C[x],D[x] are intermediate variables. The constants $r[x,y]$ are the rotation offsets, while RC[i] are the round constants. rot(W,r) is the usual bitwise cyclic shift operation, moving bit at position I

into position i+r (modulo the lane size). The constants r[x, y] are the cyclic shift offsets and are specified in the table I.

TABLE I - CONSTANTS R[X,Y] – KECCAK ALGORITHM

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	25	39	3	10	43
$y=1$	55	20	36	44	6
$y=0$	28	27	0	1	62
$y=4$	56	14	18	2	61
$y=3$	21	8	41	45	15

The constants RC[i] (see Table II) are the round constants. The following table specifies their values in hexadecimal notation for lane size 64. For smaller sizes they must be truncated.

TABLE II - CONSTANTS RC[I]- – KECCAK ALGORITHM

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

The keccak first start with the description of Keccak-f in the pseudo-code below. The number of rounds nr depends on the permutation width, and is given by nr = 12+2l, where 2l = w. This gives 24 rounds for Keccak-f[1600].

```
Round[b](A,RC) {
θ step
C[x] = A[x,0] xor A[x,1] xor A[x,2]
       xor A[x,3] xor A[x,4],
D[x] = C[x-1] xor rot(C[x+1],1),
A[x,y] = A[x,y] xor D[x],
ρ and π steps
B[y,2*x+3*y] = rot(A[x,y], r[x,y]),
χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y])
and B[x+2,y]),
ι step
A[0,0] = A[0,0] xor RC
return A
}
```

The four steps (Θ,ρπ,χ,ι) of hash function keccak have data dependency of first level, ie, the current step depends only of the outcome of the previous step. This feature allows exploring techniques of parallelism in heterogeneous systems. In this context, this paper presents a proposed architecture that exploits the parallelism using OpenCL.

5 Keccak Implementations

Pierre-Louis Cayrel[11] present an implementation of the Keccak hash function family on graphics cards, using NVIDIA's CUDA framework. That implementation allows to choose one function out of the hash function family and hash arbitrary documents. In addition he presents the first ready-to-use implementation of the tree mode of Keccak which is even more suitable for parallelization.

Guillaume Sevestre[12] 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 using CUDA.

In your work Xu Guo[10] describe a consistent and systematic approach to move a SHA-3 hardware benchmark process from FPGA prototyping to ASIC implementation, and we present our latest results for ASIC evaluation of the 14 second round SHA-3 candidates.

Perreira [13] present an keccak's implementation on FPGA using pipeline architecture with intuit to obtain performance data.

TABLE III. KECCAK'S IMPLEMENTATIONS

Authors	Title	Implementation
[11]	GPU Implementation of the Keccak Hash Function Family	NVIDIA GTX 295 GPU
[12]	Implementation of Keccak hash function in Tree mode on Nvidia GPU	Core i5-750 2.6 Ghz Nvidia GTS 250
[13]	Pipeline architecture	Virtex 5
[10]	Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC implementation	FPGA implementation
		ASIC implementation

6 Keccak on OpenCL

In this section the approach to the parallelization of Keccak will be presented. We made two implementations to try to reduce the time needed to the hash computation by simultaneously execution the keccak's algorithm. The first implementation, the host program was written in python and to execute the kernel we utilized a unique work-group with the same size of NDRange specified where all work-items in

the NDRange space computate the keccak's algorithm. The second implementation we written the host program in C language to make some tests with AMD CodeXL, and the NDRange space was divided in work-groups of 256 work-items, than we compare if has any difference between C and python's implementation.

The original Keccak structure have been almost completely maintained in this solution, even thought some adjustments have been made to maximize the performance on GPU.

The OpenCL architecture supports thousands of work-items in hardware. The host program of our implementation was written in python and kernel function on OpenCL. We utilize different sizes of NDRange and use all work-items in the NDRange to execute the four steps (Θ,ρπ,χ,ι) of keccak algorithm. To execute the tests we started with 25 work-items executed se same round of keccak and ended with 1 bilion of work-items executing the algorithm. The tests on GPU was made in an AMD/ATI Radeon HD 6400M series that has 160 Stream Processing Units, and the CPU's tests was made in a Intel Core I5. To calculate the time of the execution's kernel we got the time before the submission of the kernel to execution (T1) and the time after to kernel's execution (T2) and the result of time is the difference of T2 and T1 (T2 – T1).

The figure 6.1 shows an OpenCL kernel pseudo-code to demonstrate the execution of the first test with 25 work-items. Each work-item will instantiate the kernel function and execute completely the code.

```
1  __kernel void Keccak(__global __constant uint64_t *data,
2  »       »            __global __constant uint64_t *out)
3  {
4      int id = get_global_id(0);
5
6      if(id < 25){
7         for(int i=0; i< ROUNDS; i++){
8  »          Keccak_f(data);
9         }
10        out[id] = data[id];
11     }
12 }
```

Figure 6.1: OpenCL keccak's kernel

Lines one and two shows the definition and parameters of the kernel that will be executed per all work-items. The first parameter is the input state (matrix A 5x5), and the second parameter is the out of state after keccak-f permutation. The variable id defined in line four receive the global_ID of each work-item.

Line 6 to 8 indicates the core execution of keccak but just will be executed per work-items that have id less than 25. Finally line 10 represents the attribution of the variable out that will receive the result of keccak permutation and will be transfered to the host program.

Table IV shows the python's implementation with the numbers of work-items and the time that all work-items led to execute the algorithm. The results was compared with GPU and CPU execution.

TABLE IV. KECCAK'S IMPLEMENTATION IN PYTHON + OPENCL

No. Work-items	**Time in seconds**	
	CPU Intel core I5	GPU AMD Radeon HD 6400M
25	0.0001890659332	0.0013608932495
50	0.0002439022064	0.0007479190826
100	0.0002799034118	0.0017559528350
500	0.0008549690259	0.0007867813110
1000	0.0019378662110	0.0018019676208
50000	0.0698390007019	0.0070748329163
100000	0.130648136139	0.0138649940491
500000	0.6393702030	0.06292104721
1000000	1.29261088371	0.123764038086
50000000	62.931710	6.18758797
100000000	125.824690104	12.0365948677
500000000	628.15016818	60.4733588
1000000000	1258.75649595	119.857429981

The results of this first implementation shows that GPU execution is approximately 10 times faster than CPU execution.

AMD CodeXL is a comprehensive tool suite that enables developers to harness the benefits of AMD CPUs, GPUs and APUs. It includes powerful GPU debugging, comprehensive GPU and CPU profiling, and static OpenCL kernel analysis capabilities, enhancing accessibility for software developers to enter the era of heterogeneous computing. AMD CodeXL is available both as a Visual Studio extension and a standalone user interface application for Windows and Linux[14].

To make some tests with CodeXL we have to written the host program to C language and we make some changes in the kernel to collect more informations of the execution.

Figure 6.2 shows details of the kernel execution, and some additional information such as, duration of kernel's execution, global size and local size, kernel occupancy and others. the results were collected with AMD CodeXL.

Name	=	keccak_f
Device	=	Caicos
Command Type	=	CL_COMMAND_NDRANGE_KERNEL
Queued Time	=	192.946 millisecond
Submit Time	=	193.000 millisecond
Start Time	=	194.044 millisecond
End Time	=	223.261 millisecond
Duration	=	29.217 milliseconds
clEnqueue API Name	=	clEnqueueNDRangeKernel
clEnqueue API Start Time	=	192.909 millisecond
clEnqueue API End Time	=	192.985 millisecond
clEnqueue API Duration	=	75.401 microseconds
clEnqueue Call Index	=	27
Global Work Size	=	{256000}
Local Work Size	=	{256}
Kernel Occupancy	=	12.50%

Figure 6.2: Results colected with CodeXL.

The C implementation shows the same results of the python's implementation, the GPU execution is approximately 10 times faster than CPU execution. The table V shows some results of C implementation.

TABLE V. KECCAK'S IMPLEMENTATION IN C + OPENCL

No. work-items	**Time in seconds**	
	CPU Intel core I5	GPU AMD Radeon HD 6400M
2560	0.00544497	0.00247133
256000	0.343549	0.0292091
256000000	295.781	29.3051

7 Conclusions

This article presented an overview on the use of GPU to accelerate processing algorithms dedicated as keccak. Were presented CUDA and OpenCL platforms and a study showing that OpenCL is improving with each generation.

In the sequence was described the main module of architecture OpenCL and structure of the keccak algorithm. Keccak implementations on different technologies were presented. This algorithm is in evidence, as was recently selected as the new standard SHA-3 hash functions.

The objective of this work was not to develop the best implementation of keccak in GPU, but the use of OpenCL as an alternative for high performance applications.

For this, two implementations were coded. The first implementation, the host program was written in python and the second in C language to make some tests with AMD CodeXL: a comprehensive tool suite that enables developers to harness the benefits of AMD CPUs, GPUs and APUs.

The results shows a speedup of approximately 10 times between the CPU and GPU implementation. This gain can be further enhanced with other techniques of parallelism, such as pipeline and distribution of items running on tree model. However the aim was achieved showing that a basic implementation can achieve good level of performance.

8 References

[1] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," IEEE Micro, vol. 28, pp. 13–27, July 2008.

[2] J. Nickolls and W. J. Dally, "The GPU Computing Era," IEEE Micro, vol. 30, pp. 56–69, March 2010. 2. Oxford: Clarendon, 1892, pp.68-73.

[3] Jianbin Fang, Ana Lucia Varbanescu and Henk Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL", International Conference on Parallel Processing , 2011

[4] Khronos OpenCL Working Group, The OpenCL Specification, 2011.

[5] Aaftab Munshi, et al. OpenCL Programming Guide, 2011.

[6]FIPS 180-3, Secure Hash Standard, Cryptographic Hash Algorithm Competition, , available from http://csrc.nist.gov/groups/ST/ hash/sha-3/index.html, 2011

[7] Daemen, J. et al. "Sponge Functions". 2011, available from http://sponge.noekeon.org/Sponge Functions.pdf

[8] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, The Keccak reference, 2011.

[9] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, The Keccak SHA-3 submission, 2011.

[10] X. Guo, S. Huang, L. Nazhandali and P. Schaumont, Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations, Second SHA-3 Candidate Conference, 2010

[11] Pierre-Louis Cayrel, Gerhard Hoffmann, Michael Schneider, GPU Implementation of the Keccak Hash Function Family, SERSC International Journal of Security and Its Applications Vol. 5 No 4, October, 2011.

[12] Guillaune Sevestre, Implementation of Keccak hash function in Tree mode on Nvidia GPU, 2011

[13] Pereira, F. D. ; Ordonez, E. D. M.; Sakai, I. D. Hash function keccak: exploring parallelism with pipeline. In: PDCS- Parallel and Distributed Computing and Systems, 2011.

[14] AMD Developer Central, AMD CodeXL: comprehensive debugging profiling and analysis tool for CPU, GPU and APU. Available from http://developer.amd.com/tools/heterogeneous-computing/codexl, 2012

[15] Michael Feldman, OpenCL Gains Ground On CUDA, available from: http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html