2010 International Conference on Computational Science and Its Applications

The AES implantation based on OpenCL for multi/many core architecture

Osvaldo Gervasi, Diego Russo
Dept. of Mathematics and Computer Science
University of Perugia
Via Vanvitelli,1 06123 Perugia, Italy
osvaldo@unipg.it, diegor.it@gmail.com

Flavio Vella
Istituto Nazionale di Fisica Nucleare
Università degli Studi di Perugia
Perugia, Italy
flavio.vella@pg.infn.it

***Abstract*—In this article we present a study on an implementation, named clAES, of the symmetric key cryptography algorithm Advanced Encryption Standard (AES) using the Open Computing Language (OpenCL) emerging standard. We will show a comparison of the results obtained benchmarking clAES on various multi/many core architectures. We will also introduce the basic concepts of AES and OpenCL in order to describe the details of clAES implementation.**

This study represents a first step in a broader project which final goal is to develop a full OpenSSL library implementation on heterogeneous computing devices such as multi-core CPUs and GPUs.

***Keywords*-AES; GPU Computing; cryptography; multi/many core architectures; parallel computing.**

I. Introduction

Recently, modern GPUs (Graphical Processing Units), with their high number of parallel stream processors, are more and more used for general purpose applications that require a considerable computational power. In particular, the adoption of GPU devices in cryptographic applications is very important [1]. This domain is known as GPU computing.

The major vendors of graphic processors are releasing hardware products and programming frameworks that enable the development of applications able to fully exploit the GPU computational power.

However applications developed with these frameworks on such hardware devices are not easily portable among different architectures. This problem should be overcome by applications that comply to the Open Computing Language (OpenCL) emerging standard [2], which vendors have committed to foster.

OpenCL is the first open royalty-free standard for parallel cross-platform programming of modern processors such as GPUs and many-core CPUs. OpenCL is defined and supported by the Khronos Group[1], a consortium jointly held by the largest companies in the industry and many other promoters, supporters and members of academia. The Kronos Group purpose is to define open standards for the use and proliferation of multiplatform parallel computing.

[1]http://www.khronos.org/

The definition of version 1.0 of the standard has been published in February 2009 and GPU vendors have already released OpenCL compatible driver and development environments for their products.

II. AES

A big role in current encryption communications and security panorama is played by the Advanced Encryption Standard (AES) algorithm [3]. AES is a valid instrument to encrypt data in applications ranging from personal to highly confidential domains. The algorithm is developed by Joan Daemen and Vincent Rijmen that submitted it to AES selection process with "Rindael" codename (derived by inventors' name) [4]. Due to its characteristics, it can greatly benefit from a parallel implementation [5] and in particular from a GPU implementation [6].

As a matter of fact AES operates on data split in 4x4 matrix of bytes, called the state, and uses a symmetric key of size 128, 192, or 256 bits. The algorithm consists in several execution cycles that convert the original data into encrypted/decrypted data. Each cycle consists of several steps, including the first one, which depends on the key length. The following algorithm is used to *encrypt* data:

- KeyExpansion using *Rijndael's key schedule*
- Add the *First* round key to the state before starting the rounds with *AddRoundKey* when every byte of the state is combined with the *round key*
- First Round to n-1 Rounds
  1) *SubBytes* — on linear substitution where every byte is replaced by another one
  2) *ShiftRows* — transposition where every row of the state is cyclically shifted in a number of steps
  3) *MixColumns* — mixes the columns of the state
  4) *AddRoundKey* — every byte of the state is combined with the *round key*
- Final Round (no MixColumns)
  1) *SubBytes*
  2) *ShiftRows*
  3) *AddRoundKey*

The algorithm used to *decrypt* data is the following:

- KeyExpansion using *Rijndael's key schedule*

978-0-7695-3999-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ICCSA.2010.44

- Add the *Last* round key to the state before starting the rounds with *AddRoundKey* when every byte of the state is combined with the *round key*
- Final Round down to second round
  1) *InvShiftRows* — transposition where every row of the state is cyclically shifted in a number of steps
  2) *InvSubBytes* — on linear substitution where every byte is replaced by another one
  3) *AddRoundKey* — every byte of the state is combined with the *round key*
  4) *InvMixColumns* — mixes the columns of the state
- First Round (no InvMixColumns)
  1) *InvShiftRows*
  2) *InvSubBytes*
  3) *AddRoundKey*

AES uses the following structures:

- *Forward S-box*: is generated by determining the multiplicative inverse for a given number in Rijndael's finite field. The multiplicative inverse is then transformed using an affine transformation. Forward S-box is used to encrypt data.
- *Inverse S-box*: is simply the S-box run in reverse. First it is calculated the inverse affine transformation of the input value and then the multiplicative inverse. Inverse S-box is used to decrypt data.
- *RCON*: The round constant word array. It is used by KeyExpansion and it is the exponentiation of 2 to a user-specified value t performed in Rijndael's finite field.

For complete details about AES refer to references [3] and [4].

III. OpenCL base concept

OpenCL is an open standard aimed at providing a programming environment suitable to access heterogeneous architectures. In particular OpenCL allows to execute computational programs on multi many-core processors. Considering the increasing availability of such types of processors, OpenCL is playing a crucial role to enable the wide access of portable applications to innovative computational resources. To achieve this aim, various levels of abstraction have been introduced in the OpenCL model. OpenCL can be seen as a hierarchical collection of models from close-to-metal layer to the higher layer. The OpenCL 1.0 specification is made up of three main layer:

- *Platform:* the platform layer performs an abstraction of the number and type of computing devices in the hardware platform. At this level are made available to the developer the routines to query and to manage the computing device, to create the contexts and work-queues to submit instructions on it (it is called *kernel*).
- *Execution:* is based on the concept of kernel. The kernel is a collection of instructions that are executed on the computing device,GPU or CPU multicore, called OpenCL device. The execution of OpenCL applications can be divided in two parts: *host program* and *kernel program*. The host program is executed on CPU, defines the context (platform layer) for the kernels and manages their execution. Citing the OpenCL specification *"... when a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code on distinguished data. ..."* [7]. This definition highlights a very important concept: the higher number of cores (of CPU or GPU) the greater number of work-items will be executed in parallel. Work-items are organized into work-groups providing a more coarse-grained decomposition of the index space.
- *Language:* the language specification describes the syntax and programming interface for writing compute kernels (set of instructions to execute on computing device such as GPUs and multi-core CPUs). The language used is based on a subset of ISO standard C99 [8].

To understand in depth OpenCL, we have to analyze its memory model. The figure 1 shows the OpenCL device architecture, with memory regions and how they are related to the platform model.

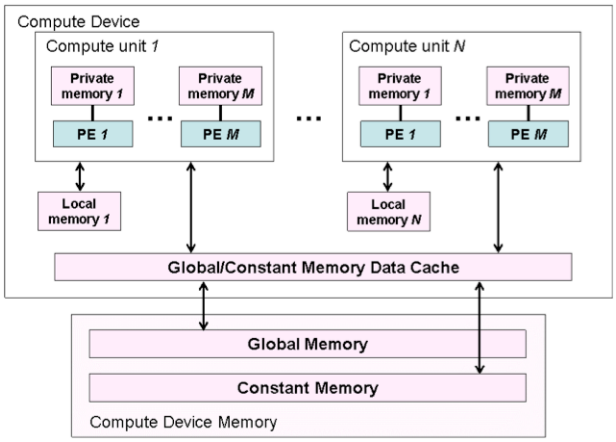

Figure 1. OpenCL device architecture with memory scopes

As you can see, there are four memory level:

- *Global Memory*: permits read/write to all work-items in all work-groups. It is advisable to minimize transfers to and from this type of memory.
- *Constant Memory*: global memory's section that is immutable during kernel execution.
- *Local Memory*: permits read/write to all work-items owned by a work-group. Can be used to allocate variables shared by in a work-group.

- *Private Memory*: permits read/write to a work-item. Variables into private memory are not visible to another work-items.

The application running on the host uses the OpenCL API to create memory objects in global memory, and to enqueue memory commands that operate on these memory objects. Besides you can synchronize enqueue command by command-enqueue barrier or using context's events.

IV. CLAES IMPLEMENTATION

Here is reported the high level description of our implementation. All steps except the second-last one are executed in CPU mode. Instead *"Perform kernel"* is executed by the OpenCL device.

1) **Read input file (plain text or ciphered)**: read from console the file to encrypt or decrypt
2) **Read AES parameters**: across interactive input, key length, the key and the action to perform (encryption or decryption) are typed by the user
3) **Transfer memory objects to device global memory**: through OpenCL API it transfers all buffer memory objects to device global memory. Objects consist of: input file, output file, SBOX or RSBOX (it depends by the action) and others memory buffers.
4) **Key expansion**: this step is executed only one time for execution. It takes the key and with *Rijndael's key schedule* expands short key into a number of separate round keys.
5) **Perform kernel**: this step is executed by OpenCL device. Some sub-operations are included in a unique kernel that uses the shared memory for optimal execution to minimize the read/write latency from/to global memory. The Shared memory is used like a memory buffer, where you can store intermediates states. As soon as the last state is computed, this buffer is copied into the global memory.
   - **SubBytes**: each byte is update using a SBOX
   - **ShiftRows**: each row is shifted by a certain offset
   - **MixColumns**: each column is multiplied with a fix polynomial *c(x)*
   - **AddRoundKey**: the subkey is combined with the state with a XOR

   The above steps are serial functions called by the kernel. The kernel uses data-parallel paradigm executing a lot of operations in multiple data domains. In fact, the input file is splitted into multiple parts depending on the local work size (number of work-item for work-group). Each Compute Unit of the OpenCL device processes the workload in a parallel way.
6) **Transfer memory objects from device global memory**: when the kernel finishes its computation, in the global memory there is an area with output data. Through OpenCL API it transfers this area from OpenCL device to host's RAM and eventually into the hard disk or other mass storage.

An important aspect of our approach is the use of *uchar4* data type. Vector literals can be used to create vectors from a set of scalars, or vectors. In this case we have used a vector of unsigned char that counts of 4 elements. This is due to the size of the state that is a matrix 4x4. With a single variable it addresses a single row of the state. Following this philosophy you can view input file like a big array of uchar4 where each element is a row. Besides, two uchar4 memory buffer are used in shared memory to store transitional states. An example of uchar4 is:

```
uchar4 test;
test.x = 'a';
test.y = 'b';
test.z = 'c';
test.w = 'd';

--
test = (uchar4)('a','b','c','d');
```

For example the function *ShiftRows* is implemented as follows, where *row* is a row of the state and *j* is the row number:

```
uchar4
shiftRows(uchar4 row, unsigned int j)
{
    uchar4 r = row;
    for(uint i=0; i < j; ++i)
    {
        r = r.yzwx;
    }
    return r;
}
```

When we operate with the columns of the state we should have synchronized all rows of the state. For this reason we have to use a lock to synchronize the rows before executing the *MixColumns* function.

V. CLAES PERFORMANCE

In this section we describe the hardware used for testing the performance of the AES algorithm considering the serial [9] and the OpenCL parallel implementation.

A. Hardware description

clAES has been tested on two video card technologies, the ATI Firestream 9270 and the Nvidia GeForce 8600 GT using the two vendors' implementation of the OpenCL driver [10], [11], and on a CPU Intel Duo E8500 device with the AMD OpenCL driver. The output of the OpenCL query on the three devices, the ATI FireStream 9270, the CPU Intel Duo E8500, and the Nvidia GeForce 8600 GT, is the following:

```
Device ATI RV770
CL_DEVICE_TYPE: CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS: 10
CL_DEVICE_MAX_WORK_ITEM_SIZES: 256 / 256 / 256
CL_DEVICE_MAX_WORK_GROUP_SIZE: 256
CL_DEVICE_MAX_CLOCK_FREQUENCY: 750 MHz
CL_DEVICE_IMAGE_SUPPORT: 0
CL_DEVICE_GLOBAL_MEM_SIZE: 512 MByte
CL_DEVICE_LOCAL_MEM_SIZE: 16 KByte
CL_DEVICE_MAX_MEM_ALLOC_SIZE: 256 MByte
```

```
CL_DEVICE_QUEUE_PROPERTIES: CL_QUEUE_PROFILING_ENABLE


Device Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz
CL_DEVICE_TYPE: CL_DEVICE_TYPE_CPU
CL_DEVICE_MAX_COMPUTE_UNITS: 2
CL_DEVICE_MAX_WORK_ITEM_SIZES: 1024 / 1024 / 1024
CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024
CL_DEVICE_MAX_CLOCK_FREQUENCY: 3166 MHz
CL_DEVICE_IMAGE_SUPPORT: 0
CL_DEVICE_GLOBAL_MEM_SIZE: 1024 MByte
CL_DEVICE_LOCAL_MEM_SIZE: 32 KByte
CL_DEVICE_MAX_MEM_ALLOC_SIZE: 512 MByte
CL_DEVICE_QUEUE_PROPERTIES: CL_QUEUE_PROFILING_ENABLE


Device GeForce 8600 GT
CL_DEVICE_TYPE: CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS: 4
CL_DEVICE_MAX_WORK_ITEM_SIZES: 512 / 512 / 64
CL_DEVICE_MAX_WORK_GROUP_SIZE: 512
CL_DEVICE_MAX_CLOCK_FREQUENCY: 1188 MHz
CL_DEVICE_IMAGE_SUPPORT: 1
CL_DEVICE_GLOBAL_MEM_SIZE: 255 MByte
CL_DEVICE_LOCAL_MEM_SIZE: 16 KByte
CL_DEVICE_QUEUE_PROPERTIES: CL_QUEUE_PROFILING_ENABLE
```

Considering the reported hardware information, it is important to underline the following aspects:

- CL_DEVICE_MAX_COMPUTE_UNIT: this value represents the number of computing devices available; respectively ten for the ATI Firestream 9270 and two for the CPU Intel E8500.
- CL_DEVICE_MAX_MEM_ALLOC_SIZE: represents the maximum dimension of the allocatable memory space for a single variable. The ATI Firestream 9270 value, 256, limits in such platform the clAES encryption to file size up to 256 Megabytes.
- CL_QUEUE_PROFILING_ENABLE: enabling this flag is possible to evaluate the execution time of clAES in the kernel program and the time of transfer of data from/to global memory (space of memory in VRAM of GPU device or space of memory reservation in RAM for CPU device). This flag has been activated to calculate the values reported in section V-B.

B. Performance tests

In this section we will present the performance tests of clAES made on three different devices using a key size of 192 bit. In figure 2 are plotted the kernel execution times (in ms) of the ATI Firestream 9270 (green curve) and of the CPU Intel E8500. The study of the CPU Intel E8500 has been performed executing the serial implementation of clAES (orange curve) and the OpenCL implementation of clAES to exploit the many core property of such CPU (blue curve).

We observe that considering a file size of 128B the execution time on the ATI Firestream 9270 is higher than on the CPU (serial or OpenCL implementations), because of the cost we need to pay to access the GPU device. Considering a file size of 256B, the measured kernel execution time on the ATI Firestream 9270 is lower than that measured on the

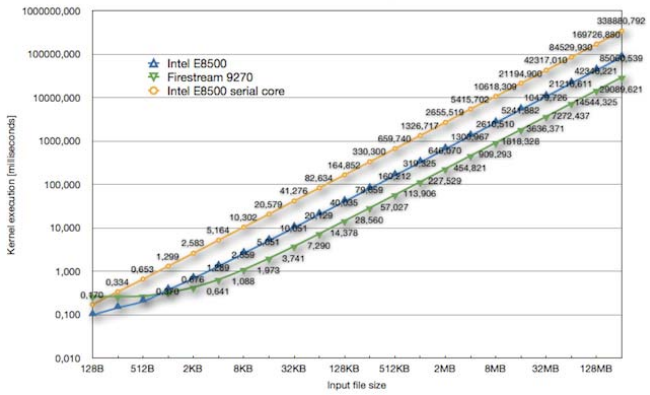

Figure 2. Plot of the kernel execution times on the ATI Firestream 9270 (green curve), on the CPU Intel E8500 Dual Core using the OpenCL implementation (blue curve) and on the same CPU using the serial implementation (orange curve) as a function of the size of the file to encrypt.

CPU executing the serial implementation, and higher than that measured in the CPU with the OpenCL implementation. From file size of 1KB the ATI Firestream 9270 is the fastest device, with a progressive increase in performance with the input file size.

In Figures 3 and 4 the kernel execution times are reported for small input file sizes (up to 2KB), respectively, without counting the time needed to transfer the data in the GPU memory and including it. We can observe that the time necessary to copy the input data to the GPU memory plays a role, but it is not the main reason that determines the higher execution times observed for the OpenCL GPU implementation for small input file sizes.

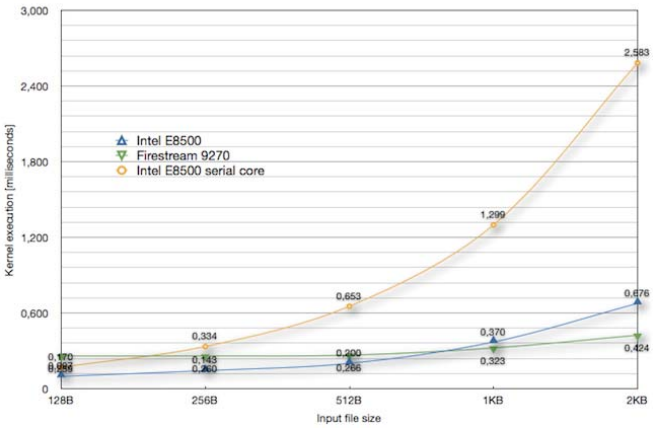

Figure 3. Plot of the kernel execution times on the ATI Firestream 9270 (green curve), on the CPU Intel E8500 Dual Core using the OpenCL implementation (blue curve) and on the same CPU using the serial implementation (orange curve) for input file sizes in the range 128 – 2048 Bytes, ignoring the time required to copy the input (output) data into (from) the OpenCL device global memory.

We can observe that the transfer of the input data in the GPU memory influences the total time measured for the kernel execution and determines an higher value of the input file size to which the curve of the execution times in the GPU (green curve) crosses those related to the execution times in the CPU, considering the serial (orange curve) and the parallel (green curve) implementations of the AES algorithm.

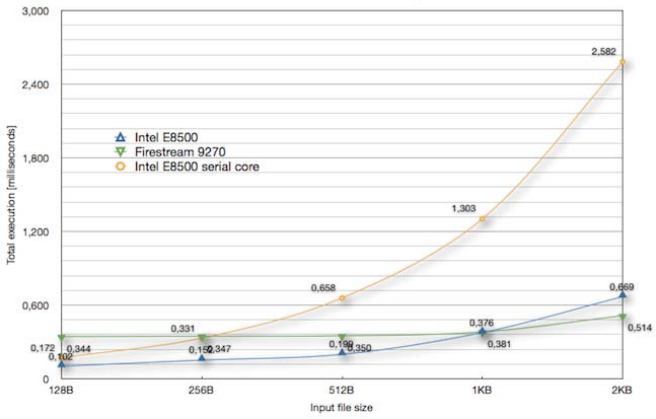

Figure 4. Plot of the kernel execution times on the ATI Firestream 9270 (green curve), on the CPU Intel E8500 Dual Core using the OpenCL implementation (blue curve) and on the same CPU using the serial implementation (orange curve) for input file sizes in the range 128 – 2048 Bytes, including the time required to copy the input (output) data into (from) the OpenCL device global memory.

We have performed some tests also using the graphic device Nvidia 8600GT.

In figure 5 are plotted the kernel execution times on two graphic processors: the Nvidia GeForce 8600GT (blue curve) and the ATI Firestream 9270 (green curve).

It is important to notice that the porting of the code in the three tested devices (the Nvidia and ATI GPUs and the multicore CPU Intel) was very simple, being sufficient to change the PATH variable associated to the related OpenCL library.

Finally in figure 6 we report in a graph a comparison between the multicore CPU Intel E8500 and the GPU ATI Firestream 9270.

The three curves represent the values of the following ratios as a function of the input file size:

- *(kernel execution time on CPU Intel E8500, OpenCL parallel implementation) / (kernel execution time on GPU ATI Firestream 9270)* (blue curve): the curve shows that the ATI Firestream 9270 execution time is higher than that of multicore CPU Intel E8500 for input file size up to 1024 bytes, approximately. If we count also the time needed to copy the input data in the GPU local memory (see figure 3), the file size at which the execution times is the same in both devices, increases slightly. For larger input file sizes, the GPU performs better than the multicore CPU.
- *(kernel execution time on the CPU Intel E8500 using the serial implementation) / (kernel execution time on GPU ATI Firestream 9270)* (green curve): the curve shows that the ATI FireStream 9270 is less efficient than the CPU Intel E8500 (serial implementation) for very small input file size, because of the time required to access the GPU device. Starting from the tested input file size value of 256B, the GPU is more efficient than the CPU. Up to input file sizes of 32KB the efficiency increase dramatically with the increase of the input file size. For input files larger than 32KB, the gain in efficiency remains approximately constant.
- *(kernel execution time on the CPU Intel E8500 using the serial implementation) / (kernel execution time on CPU Intel E8500, OpenCL parallel implementation)* (orange curve): the curve shows that the OpenCL implementation is much more efficient of the serial implementation, and is really able to exploit the multi / many cores properties of the CPU. However also in such case for small valued of the input file size the gain in efficiency is increasing with the input file size, ranging from 1.85 to 4.

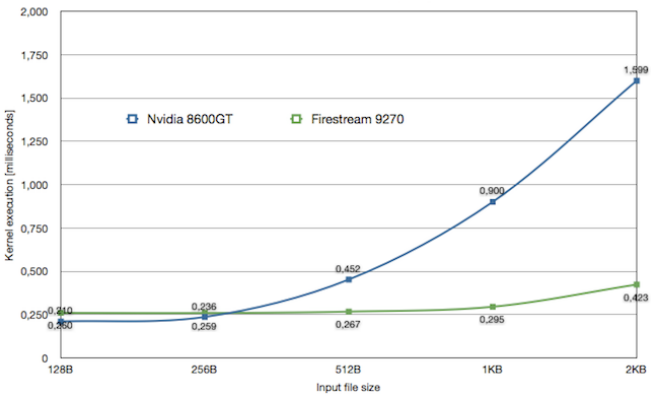

Figure 5. Kernel execution times to encrypt files which size ranges from 128 bytes to 2048 bytes for the Nvidia GeForce 8600GT (blue curve) and the ATI Firestream 9270 (green curve).

In conclusion, respect to the serial implementation of the code executed on a CPU Dual Core Intel E8500, we observe an increase in performance of a factor of 4 when considering its OpenCL implementation, demonstrating a capability of exploiting the many core CPU's property, and a factor of 11 on the ATI FireStream 9270, ignoring the time necessary for copying data in the GPU memory. We expect even a more significant improvement in performance on the ATI Firestream 9270 after having revised the OpenCL implementation and optimizing the code.

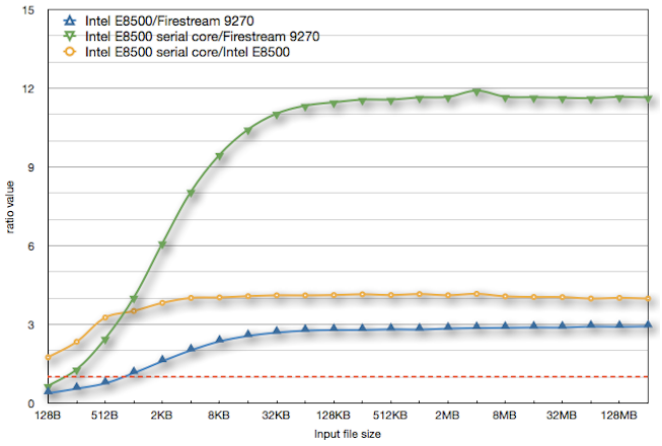

Figure 6. Comparison between the multicore CPU Intel E8500 and the GPU ATI Firestream 9270, expressed by the following ratios: *(kernel execution time on CPU Intel E8500, OpenCL parallel implementation) / (kernel execution time on GPU ATI Firestream 9270)* (blue curve), *(kernel execution time on the CPU Intel E8500 using the serial implementation) / (kernel execution time on GPU ATI Firestream 9270)* (green curve), and *(kernel execution time on the CPU Intel E8500 using the serial implementation) / (kernel execution time on CPU Intel E8500, OpenCL parallel implementation)* (orange curve).

VI. Conclusion

In this paper we presented the results of the performance tests made running an OpenCL implementation of the AES cryptographic algorithm on GPUs and on multi core CPU.

The results demonstrates the full capability of the OpenCL standard to exploit the many core property of CPUs and GPUs.

These preliminary, very good results, can lead to better performances on GPUs after a further optimization of the OpenCL code.

As for scalability and portability, we demonstrated how OpenCL applications can scale on various heterogeneous computing resources.

Our future work will involve the following activities:

- optimization of clAES code
- development of a patch for the inclusion of the OpenCL approach in OpenSSL [12]
- development of the OpenCL implementation of all cryptographic algorithms adopted in OpenSSL

We are confident that OpenCL is a promising standard for parallel programming and hopefully it will be extended to other platforms cooperating with actual parallel programming models (for example POSIX Threads, MPI) and parallel languages.

References

[1] D. L. Cook, J. Ioannidis, A. D. Keromytis, J. Luck, *CryptoGraphics: Secret Key Cryptography Using Graphics Cards*, RSA Conference, Cryptographer's Track (CT-RSA),2005.

[2] Khronos Group, *OpenCL 1.0 Specification*, http://www.khronos.org/opencl

[3] National Institute of Standards and Technology (NIST), *FIPS 197: Advanced Encryption Standard (AES)*, 1999.

[4] J. Daemen, V. Rijmen, *AES Proposal: Rijndael. Original AES Submission to NIST*, 1999. AES Processing Standards Publications, http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[5] N. Sklavos, O. Koufopavlou, *Architectures and VLSI Implementations of the AES-Proposal Rijndael*, IEEE Transactions on Computers, Vol. 51, Issue 12, pp. 1454-1459,2002.

[6] Svetlin A. Manavski, *CUDA compatible GPU as an efficient hardware accelerator for AES cryptography*, In Proc. IEEE International Conference on Signal Processing and Communication, ICSPC 2007, (Dubai, United Arab Emirates), pp.65-68, 2007.

[7] Khronos Group, *Kernel definition*,page 20 of http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

[8] The features of C99 are described in the document "Rationale for International Standard – Programming Languages – C ", Revision 5.10, available at the URL: http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf

[9] Hoozi *AES Serial implementation*, www.hoozi.com/Articles/AESEncryption.htm

[10] AMD, *ATI Stream SDK v2.0 Beta*, http://developer.amd.com/gpu/ATIStreamSDKBetaProgram/Pages/default.aspx

[11] Nvidia,*OpenCL GPU Computing Support on NVIDIA's CUDA Architecture GPUs*, http://www.nvidia.com/object/cuda_opencl.html

[12] OpenSSL Project http://www.openssl.org/