2010 First International Conference on Networking and Computing

AES encryption implementation on CUDA GPU and its analysis

Keisuke Iwai and Takakazu Kurokawa
Dept. of Computer Science
National Defense Academy of Japan
Kanagawa, Japan
Email: {iwai,kuro}@nda.ac.jp

Naoki Nisikawa
Technical Research and Development Institute
Ministry of Defense
Tokyo, Japan

***Abstract*—GPU has a good performance ratio and exhibits the capability for applications with high level of parallelism despite its inexpensive price. The support of integer and logical instructions on the latest generation of GPU makes us to implement cipher algorithms easier with the same instructions. However the decisions such as parallel processing granularity or memory allocation place imposed heavy burden on programmers. For this reason this paper shows the results of several experiments to study relation between memory allocation style of AES parameters and granularity as the parallelism exploited from AES encoding process using CUDA with NVIDIA Geforce GTX285. The result of experiments cleared up that the 16Byte/thread granularity had the highest performance and it achieved approximately 35Gbps throughput. Moreover, implementation with overlapping between processing and data transfer brought up 22.5Gbps throughput including data transfer time. Also, it cleared up that it is important to decide granularity and memory allocation to effective processing in AES encryption on GPU.**

***Keywords*-GPU; AES; Accelerator**

I. INTRODUCTION

GPU(Graphics Processing Unit) is a hardware specialized in 3D graphics processing. At the time of DirectX 9, 16-bit floating point format has been contrived as dynamic range to record enormous amount of data like real world. After that, GPU has advanced by adopting powerful 16-bit floating point architecture. The research to apply this GPU's powerful computational ability with general purpose computing has become popular since 2004. However, only fluid calculation or Newtonian N-body problem have benefited from GPU and it was very hard to implement other applications on GPU.

In response to this situation NVIDIA that is a GPU vendor has developed and released so-called "CUDA"(Compute Unified Device Architecture)[1]. Integer and logical instruction have newly been supported for GPGPU and make it easier to implement applications using these operations[2]. In addition, C-like programming language has become available, so programmers have been able to utilize GPU's computing power with relatively easy use[3]. In this way, GPGPU has continued to advance as the good cost-performance environment and has been noticed from various research fields[4].

After CUDA was released, programmers was able to certainly code easily. Nevertheless, programmers still have to consider many factors to exploit the power. In particular, there is an absolute fact that programmers have to make decisions of several parameters such as memory allocation or how to assign computations to threads.

For this reason, we implemented a variety of AES encoding with different combination of memory allocation style and parallel processing granularity. Using these results of experiments this paper discuss the effectiveness for performance about difference of memory allocation method , computation granularity and more.

II. CUDA

CUDA is a GPGPU development environment released by NVIDIA. Programmers write a thread program and are supposed to specify the number of threads and thread blocks arbitrarily. From the point of view of hardware, quite a lot of processors and on-chip memory have been assembled and thread parallelism allows the processors not to be kept idle.

A. Hardware Model

Fig. 1 illustrates the CUDA architecture. GPU chip has N multiprocessors(MP) and each MP has M scalar processors(SP), 16KB shared memory, and several 32-bit registers. On the whole, the chip constitutes a hierarchical SIMD architecture. Cutting control unit such as conditional branch component from a instruction unit allows computing unit density to increase.

Also, GPU has a large global memory which is also called Video RAM. GPU's memory access system is composed of the same hierarchical structure as the processor's structure. For this reason, threads can access the closer memory to their own location and the whole latency is hidden.

In addition, there is constant memory per MP that can have almost same latency as shared memory in the case of cache references to constant memory. In other word, data itself is located in global memory, but if cache reference happens the entity is cashed into constant cache.

In this paper, we used NVIDIA Geforce GTX285, which is equipped with 30 MPs, 8 SPs per MP, 16384 registers per MP, and a 1GB global memory.

978-0-7695-4277-5/10 $26.00 © 2010 IEEE
DOI 10.1109/IC-NC.2010.49

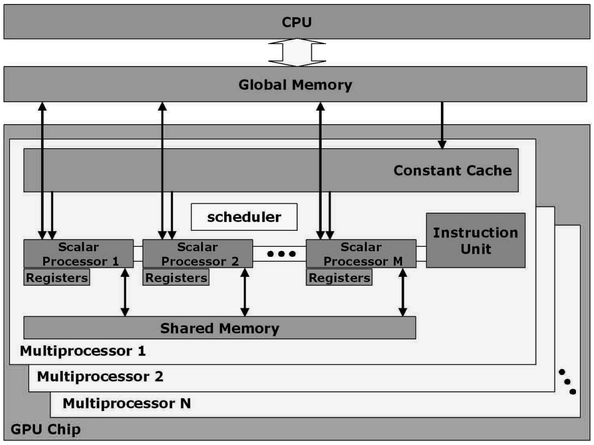

Figure 1. CUDA Architecture.

B. Software Model

For corresponding to the hardware model, thread is also composed of hierarchical SIMD architecture. The mass of thread is called thread block and programmers specify several number of thread blocks normally when a job is issued from CPU to GPU. While each thread is carried out on a SP one more thread block is assigned to a MP. MP resources such as shared memory and registers are evenly divided by the number of thread blocks. Each thread is carried out on a SP. Local variables in thread program are assigned to registers in MP.

In addition, thread assignments in MP are always executed by 32 threads and these threads are executed at the same time. This unique definition is called "warp" in CUDA as shown in Figure2. A warp is, what is called, a common destiny.

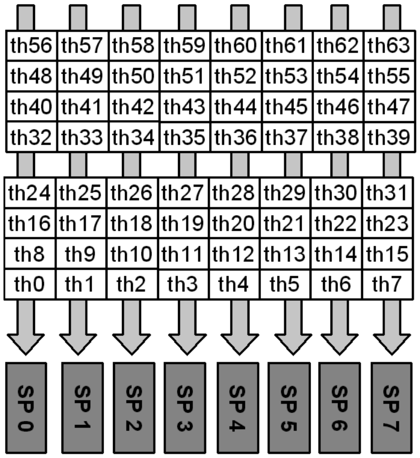
th56 th57 th58 th59 th60 th61 th62 th63
th48 th49 th50 th51 th52 th53 th54 th55
th40 th41 th42 th43 th44 th45 th46 th47
th32 th33 th34 th35 th36 th37 th38 th39
th24 th25 th26 th27 th28 th29 th30 th31
th16 th17 th18 th19 th20 th21 th22 th23
th8 th9 th10 th11 th12 th13 th14 th15
th0 th1 th2 th3 th4 th5 th6 th7
SP 0
SP 1
SP 2
SP 3
SP 4
SP 5
SP 6
SP 7

Figure 2. Warp execution in CUDA.

C. Memory access system for high memory bandwidth

While the calculation unit is 32 threads, memory access unit is 16 threads in CUDA[3].

1) Coalesced Access of global memory: Memory access cycle to global memory takes between 400 and 600 cycles and this latency is fairly low for the access speed in GPU. On the other hand, the interface between global memory and SP is by far more broad than CPU. To exploit this characteristic, memory access of global memory is hidden by issuing coalesced memory access instruction if thread access would access data with basically no stride access. CUDA compiler is in charge of whether global memory access become coalesced.

To cover this weak point, the interface between global memory and SPs is by far broader than general CPU and if threads will access with basically no stride access the CUDA compiler generates the coalesced load or store instructions at the time of compile.

2) Shared memory interleave and the avoidance of bank Conflict: Memory access to shared memory is as low latency as registers. Shared memory is divided into 16 banks(4-byte per bank). In the case of access to different bank by each thread, threads can load or store data in parallel. However, if each thread will access the same bank the memory access induces in serial. At the time the latency will be 16 times different at the maximum by how to allocate data on shared memory.

III. AES

AES is a symmetric block cipher which was introduced in 2001 by NIST[5]. AES encrypts and decrypts plaintext and ciphertext blocks using a key size of 128-bit, 192-bit or 256-bit, and the calculation unit based on encryption is 1 byte. This cipher executes the iteration of the same round, which number of iterations is different from the key size. In this paper we selected only 128-bit version, which consists of 10 rounds. Each round consists of four transformations - SubBytes, ShiftRows, MixColumns, and AddRoundKey. The final round is slightly different from other rounds and don't include MixColumns.

Our implementation based on optimized ANSI C source code for AES which is provided as a part of OpenSSL the open source toolkit for SSL/TLS[6]. Its algorithm defines round processes combined into a transformation using look up table called "T-box" and ex-or operation simply. Let a be round input which is divided into each 32bits, round output e is represented as follows:

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}] \oplus k_j \quad (1)$$

where T_0, T_1, T_2, T_3 are lookup table and k_j is the j-th column of a roundkey. This algorithm contains only 4 look up table transformations and 4 ex-or operations.

Furthermore, AES has some modes - ECB(Electric Code Book), CBC(Cipher Block Chaining), and so on. We selected ECB mode because this can exploit GPU's powerful parallel processing power easily.

IV. RELATED WORKS

Cook et al. achieved AES encryption at 1.53Mbps throughput using Open GL on NVIDIA Geforce3 Ti200[7]. However its performance potentials only 2.3% of CPU (Pentium IV 1.8GHz) performance because of earlier GPU architecture didn't have enough instruction set for general purpose computing. Also Harrison et al achieved AES encryption at 870.8 Mbps throughput on Geforce 7900GT using Direct X9[8].

Manavski implemented CUDA-AES and achieved astounding 8.28Gbps throughput rate with an input size of 8MB using NVIDIA Geforce 8800GTX[9]. Authors of this paper insist that the thread block size leads to the fastest implementation in CUDA-AES, and performance improvement can be observed as the number of thread block increases. Furthermore, authors of this paper note that the effort to reduce the usage of the shared memory is needed for more performance in CUDA-AES because the shared memory is divided into equally-sized memory modules at the time of AES encryption process. However, if bank conflict happens the access has to be serialized and causes slow down of the performance. But there aren't no argument on concrete allocation of shared memory.

Di Biagio et al. also implemented counter mode AES (AES-CTR) with CUDA using NVIDIA Geforce 8800GT[10]. Authors of this paper achieved 12.5Gbps throughput rate with an input size of 128MB considering about processing granularity. They defined as fine-grained design a solution exposing the internal parallelism of each AES rounds. They proposed four 32-bit words blocks dispatch each threads as a fine-grain processing. Moreover, coarse-grained design was defined exploiting higher-level parallelism between independent plaintext blocks which works each threads processing for each 128-bit plaintext blocks.

Nishikawa el. al. also discussed granularity in [11]. They defined one thread processing one plaintext block (which contains 16Bytes) as 16Bytes/thread and also other granularities such as 4Bytes/thread and 1Bytes/thread their definition similar to 16Bytes/thread. They implemented an AES-ECB encoding according to standard AES algorithm without T-Box which achieved 2.45Gbps with Geforce GTX285. And also they proposed DES implementation on GTX285 which aimed at brute force attack[12].

Table I shows peak performances and implementation environments of these previous works. This paper discusses more detailed implementation of AES referred these previous works respectively and try to bring out more AES performance using GTX285.

Table I
PERFORMANCE OF PREVIOUS WORKS.

Reference	Device	Language	Throughput
Cook et. al.[7]	Geforce3 Ti200	OpenGL	1.53Mbps
Harrison et. al.[8]	Geforce 7900GT	DirectX9	870.8Mbps
Manavski [9]	Geforce8800GTX	CUDA	8.28Gbps
Di Biagio et. al.[10]	Geforce8800GT	CUDA	12.5Gbps
Nishikawa et. al.[11]	Geforce GTX285	CUDA	6.25Gbps

V. AES ENCRYPTION IMPLEMENTATION ON CUDA GPU

This section discusses about a design on parallel AES ECB encoding program.

A. Granularity of parallel processing

1) 16Bytes/thread: 16Bytes/thread means an paralleling method that each thread is mapped to each plaintext block consisting of 16bytes. This implementation has an advantage that this method requires no synchronization and no shared data between threads because of the encryption process of a plaintext block does not need to process plaintext block in parallel.This granularity uses a parallelism of between plaintext blocks only.

2) 8Bytes/thread and 4Bytes/thread: 8Bytes/thread granularity processes one plaintext block with two threads. Concurrently, this method exploits parallelism between plaintext blocks. This method needs shared memory to share intermediate data by two threads and also needs synchronization. 4Bytes/thread granularity processes one plaintext block with 4 threads. This method is different from 8Bytes/thread in its number of threads for a block sharing. This method requires shared memory and synchronization in the same reason with 8Bytes/thread.

3) 1Byte/thread: It is absolutely better to process AES encoding with 32bit operating unit at least because of AES encoding algorithm using this study is optimized for 32bit processing. However, it is able to process 1Byte data using a thread respectively because of AES was designed with 8bit operation unit. 1Byte/thread means that 16threads are required to process a plaintext block. This granularity is designed to compare earlier study and also other granularities although this granularity will bring not good performance for GPU which has 32bit operation unit.

B. Memory Allocation

1) Key and T-box: T-box and round keys are read-only data and they are able to be shared between all threads. According to such behavior, these valuables match to allocate on constant memory. Even if they ware allocated on shared memory, AES would indicate good performance because of shared memory provided low latency access absolutely guaranteed too. But allocating them on shared memory wastes its capacity because shared memory is able to be shared between threads belonging in the thread blocks.

2) Plaintext: Plaintext is stored on global memory at first. When AES encoding is started, plaintext is loaded on shared memory sequentially to share intermediate data between stream processors excepted at 16Byte/thread granularity because of this granularity allocate all plaintext and intermediate values allocate on registers. We implemented two types of memory allocation patterns. As shown in Fig. 3 and Fig. 4, we can select two options about how to allocate data on memory - Array of Structure(AoS) and Structure of Array(SoA). AoS deals with plaintext as the way it is, and SoA allocates each element of plaintext into one array. They provide difference of appearance of bank conflict. To reduce bank conflict occurrences, we select better allocation pattern for each implementations. Fig. 3 and Fig. 4 also show examples of thread access pattern to shared memory in case of granularity at 8Bytes/thread.

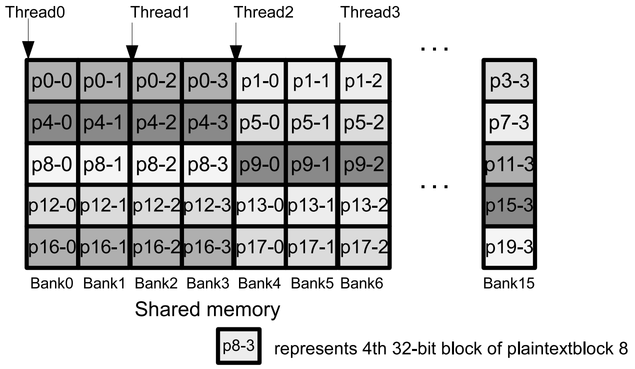
Thread0
Thread1
Thread2
Thread3
p0-0 p0-1 p0-2 p0-3 p1-0 p1-1 p1-2 ... p3-3
p4-0 p4-1 p4-2 p4-3 p5-0 p5-1 p5-2 p7-3
p8-0 p8-1 p8-2 p8-3 p9-0 p9-1 p9-2 ... p11-3
p12-0 p12-1 p12-2 p12-3 p13-0 p13-1 p13-2 p15-3
p16-0 p16-1 p16-2 p16-3 p17-0 p17-1 p17-2 p19-3
Bank0 Bank1 Bank2 Bank3 Bank4 Bank5 Bank6 Bank15
Shared memory
p8-3 represents 4th 32-bit block of plaintextblock 8

Figure 3. Array of Structure, an allocation of plaintext.

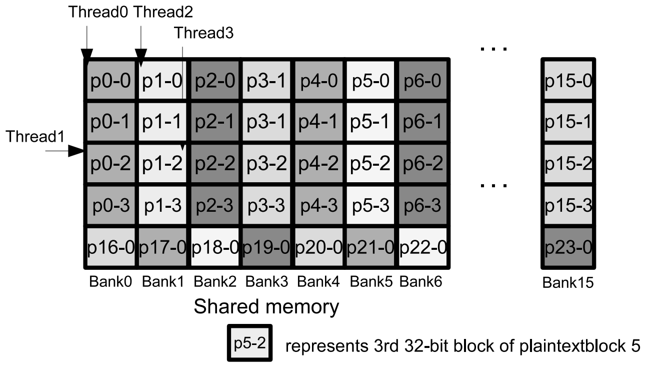
Thread0 Thread2
Thread3
Thread1
p0-0 p1-0 p2-0 p3-1 p4-0 p5-0 p6-0 ... p15-0
p0-1 p1-1 p2-1 p3-1 p4-1 p5-1 p6-1 p15-1
p0-2 p1-2 p2-2 p3-2 p4-2 p5-2 p6-2 ... p15-2
p0-3 p1-3 p2-3 p3-3 p4-3 p5-3 p6-3 p15-3
p16-0 p17-0 p18-0 p19-0 p20-0 p21-0 p22-0 p23-0
Bank0 Bank1 Bank2 Bank3 Bank4 Bank5 Bank6 Bank15
Shared memory
p5-2 represents 3rd 32-bit block of plaintextblock 5

Figure 4. Structure of Array, an allocation of plaintext.

C. T-box duplication

T-box consists of 256 entries of 32bit data. Each AES round requires four T-box transformations and T-boxes for these transformations are provided from shift operation applying one T-box. Another way of providing these four T-box transformations is using pre-computed four T-boxes. Pre-computed four T-boxes require four times much memory space than single T-box.

D. Cutdown of thread block switching

In usual CUDA applications, for massively parallel processing data is respectively mapped to each thread. For example in 3D rendering, each pixel or vertex are relatively mapped to each thread. In fluid computation, each particle are mapped to each thread. Similarly in AES, we can map each plaintext to each thread just like above applications, but the time of one encryption by threads is differently slight. Therefore, the overhead of switching thread block in AES tends to be bigger and unignorable than other applications.

For this reason, after threads finished encrypting plaintexts in charge, their threads returns to the starting point and continues to encrypt other plaintexts again. If doing so, only low number of threads can encrypt quite a few of plaintexts, and we can avoid the overhead of switching thread blocks in AES.

E. Overlapping GPU processing and memory copy

There is need to consider about overhead by data transfer between CPU and accelerator such as GPU to exploit effective performance. To hide this overhead, CUDA provides overlapping data transfer (memory copy) and processing. We implemented AES encoding process with overlapping transferring plaintext to (and ciphertext from) global memory of GPU and GPU's AES process. Fig 5 shows this overlapping.

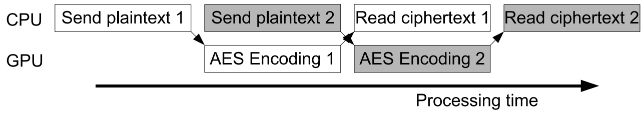

Figure 5. Overlapping data transfer and processing.

VI. EXPERIMENTAL RESULTS

A. Environment

Table II shows the specification of our computer system for this experiment.

Table II
SPECIFICATION OF COMPUTER.

CPU	Corei7 Quadi7-920(2.66GHz)
Memory	6GB
OS	CentOS5.3 (kernel ver2.6.18)
Compiler	gcc ver4.1.2(option -O3)
GPU Accelerator	NVIDIA Geforce GTX 285
GPU Memory	1GB
CUDA Compiler	nvcc ver2.3

Implemented AES is 128-bit AES encoding algorithm (ECB mode). Round key is generated by CPU once and transferred to GPU's global memory. Plain text which is made with random value is generated by CPU too. A size of plaintext is 256MB fixed size to evaluate around peak performance each implementations. Number of thread blocks is 60 and number of threads is 512 fixed of all execution.

B. Throughput

Table III shows the throughput of each implementations. At first basically conditions of each implementations are shown, because of table III contains selected data from conditions led from discussion of section V. All results shown in table III applied 4 T-boxes implementation. Preliminary experience indicated 4 T-boxes implementation achieved better performance than one T-box implementation because of less computation than one T-box. Moreover, requiring memory space for 4 T-boxes didn't effect for performance.

About plain text allocation, excepted 16Bytes/thread and 1Byte/thread, both AoS and SoA implementation ware evaluated. In case of computation granularity at 16Bytes/thread, shared memory for plaintext did not required because of registers ware able to replace shared memory. Allocating plaintext at registers achieved higher performance than allocating plaintext at shared memory because of it brought in not only no memory access conflict but also no computation for memory address computation.

As a result, one of 16Bytes/thread implementation achieved the highest 35.2 Gbps throughput. 16Byte/thread had constitute advantage compared with other implementations that they need no shared memory for processing AES encoding and also no synchronization. Requiring no shared memory means bringing in not only high speed register access but also no memory bank conflict.

About allocating place of T-box, allocating at constant memory provided critical decrees in performance. Constant memory will give high speed access with coherent memory access because constant memory equips cache system. However T-box transformation provided random access, but shared memory is able to adopt to require random memory access.

According to these experimental results, it is better to process T-box transformations using shared memory and other granularity implementations in table III didn't accept allocating T-box on constant memory.

About allocating place of round keys, there was a small deference for performance. Allocating round keys on shared memory was about 2% faster than allocating it on constant memory. Round key access required coherent memory access, that's why they have almost same throughput.

Next, let us discuss about 8Bytes/thread and 4Bytes/thread. The highest throughput was 26.9Gbps but they had almost same throughput. Their throughput are almost 30% lower than 16Bytes/thread implementations. 8 and 4Bytes/threads required shared memory to share plaintexts and intermediates. But shared memory access brings on bank conflicts. Synchronizations, shared memory access and bank conflicts are major causes of this decrease in performance.

At these granularity, allocating round keys on constant memory led better performance than allocating shared memory which is contrary result compared with 16Bytes/thread result. The reason of this difference was bank conflicts which increases when 4 and 8Bytes/thread used shared memory to store plaintext and intermediate data. To allocate round keys on constant memory led lower bank conflicts than using shared memory for round keys.

AoS plaintext allocation implementation provided about 1.5 times better performance than SoA implementation at 4byte/thread granularity. SoA implementation occurred 2 times more bank conflicts than AoS implementation because of this implementation provided 4-way bank conflict. AoS implementation, on the other hand, would provide no bank conflict without T-Box transformation. In the other case of 8Bytes/thread, difference of performance between SoA and AoS implementation was close. AoS implementation was 1.2 times faster than SoA implementation because of memory access patten between threads was not so different by each implementations.

Finally 1Byte/thread achieved very low throughput but it was no surprises because implemented algorithm was designed using 8bit operation which was made from algorithm optimized to 32bit operation divided into four operations. Results at this granularity showed different behavior from 4, 8Bytes/thread. Allocating round keys on constant memory was not valid to performance.

Considering number of thread blocks and treads is very important. All of this experiments ware done in condition under a number of thread blocks and threads ware fixed. Fact that these parameters effect to performance is well known[10] , but the case of evaluating around peak performance it is better to set large number of threads. Actually, our experiments indicated that the case of setting 512 threads per block achieved almost best performance. Although 256 threads per thread block achieved the best performance in some case, differences against 512 threads per block ware around 0.1% to 0.4%.

C. Overlapping Data transfer

Performances shown up to here are excepted data transfer time to discuss effectiveness about difference of granularity and so on. To evaluate real effectiveness provided by GPU, discussion data transfer overheads between CPU and GPGPU is important.

Our implementation results ware evaluated with AES plaintext size at 256MB. From measurement of data transfer time, there is about 0.08ms overhead which includes both copy to/from global memory from/to CPU. Throughput including this data transfer time at 16Bytes/thread(T-box on

Table III
THROUGHPUT OF EACH IMPLEMENTATION.

Granularity	16B/th	16B/th	16B/th	8B/th	8B/th	8B/th	4B/th	4B/th	4B/th	1B/th	1B/th
T-box	constant	shared	shared	shared	shared	shared	shared	shared	shared	shared	shared
Key	constant	constant	shared	constant	shared	constant	constant	shared	constant	shared	constant
Memory Allocation	NA	NA	NA	AoS	AoS	SoA	AoS	AoS	SoA	AoS	AoS
Throughput[Gbps]	5.0	34.4	35.2	26.9	23.9	23.4	25.3	25.0	17.1	2.9	2.6

shared memory and round keys on shared memory) achieved only 13.4Gbps. We implemented another AES encoding program which apply overlapping data transfer and AES processing to exploit more performance. As a result, implementation applied overlapping to 16Bytes/thread(T-box on shared and round keys on shared memory) implementation archives 22.0Gbps. This implementation divided plaintext data into 4 plaintext blocks and also AES encoding programs worked four times independently. In case of this experience, although dividing plaintext into 4 blocks achieves the best performance, there is need to more discussions about overlapping data transfer and processing.

Finally, the best throughput achieved 28.39 fold speed up compared with Core i7-920 2.66GHz CPU implementation achieved 1.2Gbps.

VII. CONCLUSION

This paper presents the analysis about the effectiveness for AES implementation from various conditions such as parallel processing granularity, memory allocation and so on.

There are over 10 times difference of performance by the kind of best GPU implementation which achieved 35.2Gbps throughput and 28.39 fold speed up compared with Core i7-920 2.66GHz CPU implementation.

We found that such implementation granularity at 16Bytes/thread tended to be effective. Moreover, common data table T-box and round keys allocating on shared memory achieves the best performance but round keys which requires coherent accesses will be able to allocate constant memory.

The best result of this paper 35.2Gbps encoding throughput strongly alleged the potential of CUDA GPU for cryptographic accelerator. As high end application, there is a possibility for such as code breaker but we expect a small accelerator for a block cipher accelerator because of GPU equipped most of computers such as notebook PC too.

Additionally, we would like to apply our AES implementation methods to another common key block ciphers such as MISTY.

REFERENCES

[1] (2009) NVIDIA CUDA. [Online]. Available: http://developer.nvidia.com/object/cuda.html

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008.

[3] *NVIDIA CUDA Programming Guide 2.3*, NVIDIA Corp., 2009.

[4] (2009) NVIDIA CUDAZONE. [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[5] *FIPS-197 Advanced Encryption Standard(AES)*, National Institute of Standards and Technology (NIST), 2001.

[6] T. O. Project. (2007) Openssl: The open source toolkit for ssl/tls. [Online]. Available: http://www.openssl.org

[7] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, "Cryptographics: Secret key cryptography using graphics cards," in *In RSA Conference, Cryptographer !𝕤 Track (CT-RSA*, 2005, pp. 334–350.

[8] O. Harrison and J. Waldron, "Aes encryption implementation and analysis on commodity graphics processing units," in *9th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007, pp. 209–226.

[9] S. A. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in *IEEE International Conference on Signal Processing and Communication, ICSPC*, 2007, pp. 65–66.

[10] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi, "Design of a parallel aes for graphics hardware using the cuda framework," *Parallel and Distributed Processing Symposium, International*, pp. 1–8, 2009.

[11] N. Nishikawa, K. Iwai, and T. Kurokawa, "Granularity optimization method for aes encryption implementation on cuda (in Japanese)," in *IEICE technical report. VLSI Design Technologies (VLD2009-69)*, Kanagawa, Japan, Jan. 2010, pp. 107–112.

[12] ——, "Acceleration of the key crack against cipher algorithm using cuda (in Japanese)," in *IEICE technical report. Computer systems 109(168)*, Sendai, Japan, Jul. 2009, pp. 49–54.