International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 2, Number 1, pages 131–145, January 2012

Acceleration of AES encryption on CUDA GPU

Keisuke Iwai
Department of Computer Science, National Defense Academy
1-10-20 Hashirimizu Yokosuka Kanagawa 239-8686, Japan

Naoki Nishikawa
Technical Research and Development Institute, Ministry of Defense
2-2-1 Nakameguro Meguroku Tokyo 153-8630, Japan

Takakazu Kurokawa
Department of Computer Science, National Defense Academy
1-10-20 Hashirimizu Yokosuka Kanagawa 239-8686, Japan

Received: January 31, 2011
Revised: May 20, 2011
Revised: July 4, 2011
Accepted: September 21, 2011
Communicated by Yasuaki Ito

Abstract

GPU exhibits the capability for applications with a high level of parallelism despite its low cost. The support of integer and logical instructions by the latest generation of GPUs enables us to implement cipher algorithms more easily. However, decisions such as parallel processing granularity and memory allocation impose a heavy burden on programmers. Therefore, this paper presents results of several experiments that were conducted to elucidate the relation between memory allocation styles of variables of AES and granularity as the parallelism exploited from AES encoding processes using CUDA with an NVIDIA GeForce GTX285 (Nvidia Corp.). Results of these experiments showed that the 16 bytes/thread granularity had the highest performance. It achieved approximately 35 Gbps throughput. It also exhibited differences of memory allocation and granularity effects around 2%–30% for performance in standard implementation. It shows that the decision of granularity and memory allocation is the most important factor for effective processing in AES encryption on GPU. Moreover, implementation with overlapping between processing and data transfer yielded 22.5 Gbps throughput including the data transfer time.

Keywords: GPGPU, AES, Accelerator

1 Introduction

The Graphics Processing Unit (GPU) is hardware that is specialized in 3D graphics processing. At the generation of DirectX9 (Microsoft Corp.), 16–bit floating point as a surface format was contrived as the dynamic range to record enormous amounts of data, just as in the real world. Subsequently, GPUs advanced by adopting powerful 16–bit floating point architecture. Research undertaken to apply this powerful GPU computational ability with general purpose computing, called GPGPU, has become popular since 2004. However, only fluid calculations or Newtonian N-body problems benefited from use of the GPU. It has remained very difficult to implement other applications on a GPU.

In response to these circumstances, Nvidia Corp., a GPU vendor, developed and released "Compute Unified Device Architecture " (CUDA) [9]. The integer and logical instructions have been newly supported for GPGPU. It has become easier to implement applications using these operations[2]. In addition, C-like programming language has become available, so programmers have been able to use GPU's computing power with ease[11]. In this way, GPGPU has continued to advance a good cost-performance environment and has attracted attention from various research fields[10].

The fact that CUDA had began to contrive integer and logical operations provided a new field for logical application such as cipher algorithms. We are also interested in accelerating cipher algorithms as non-numerical applications by CUDA. The Advanced Encryption Standard (AES) [6] is used widely as the standard cipher algorithm today. In fact, AES can be parallelized with ease in cases of ECB or CTR mode. The effectiveness for performance has not been good, although some research toward accelerating cipher processing with GPU was reported before CUDA was released. After CUDA was released, programmers were able to implement code and thereby obtain more accelerating performance. Nevertheless, programmers must still consider numerous factors to exploit power, including cipher applications. Particularly, problems such as optimizing the way of using memory of CUDA of various kind and considering computation granularity trouble many programmers.

To tackle these problems, this paper presents many types of implementation of AES, changing memory allocation and computation granularity with CUDA to determine the best programming style. The rest of this paper is organized as follows. Section 2 introduces the CUDA architecture briefly. Section 3 overviews optimized software AES implementation and its algorithm. Related works are explained in Section 4. Section 5 presents the proposed AES implementation patterns. In Section 6, throughput results of implementation and discussions of effectiveness of each implementation are presented. Finally, Section 7 describes our conclusions.

2 CUDA

CUDA is a GPU development environment, as is GPGPU released by Nvidia Corp. Programmers write a threaded program. They must then specify the number of threads and thread blocks arbitrarily. From the point of view of hardware, numerous processors and on-chip memory have been assembled. Thread parallelism eliminates idling of the processors.

2.1 Hardware model

Figure 1 portrays the CUDA architecture. The GPU chip has N $\times$ multiprocessors (MP), and each MP has M $\times$ scalar processors (SP), 16 KB shared memory, several 32–bit registers and a shared instruction cache. Overall, the chip constitutes a hierarchical SIMD architecture. Cutting control units such as conditional branch components from an instruction unit increases the computing unit density.

2.2 Memory organization

Global memory The GPU has a global memory called video RAM, which constitutes the largest memory area in the GPU. Global memory is used to communicate between the CPU and GPU because it can be accessed by both the CPU and all SPs. Although each SP is able to access directly any data loaded on the global memory, data will be transported on shared memory and registers at the beginning of processing in many cases because the latency of global memory is much higher.

Shared memory Shared memory is used on each MP. SPs located on the same MP are able to access the same shared memory. Shared memory provides as much fast memory access as registers do.

Registers The register is the fastest memory storage of GPU. The register can be accessed by fixed SPs such as registers for classical general purpose processors.

Constant memory Constant memory is the fixed cacheable area of global memory. Each MP equips the constant cache, which caches the data of constant memory. Contents of constant memory cannot be written/modified by the GPU. Constant memory is stored as read-only data and is shared with all SPs. Although the cache functions well, the access latency of constant memory will be as fast as the registers.

As described in this paper, we used an NVIDIA GeForce GTX285 (Nvidia Corp.) equipped with 30 MPs, 8 SPs per MP, 16384 registers per MP, and 1 GB global memory.

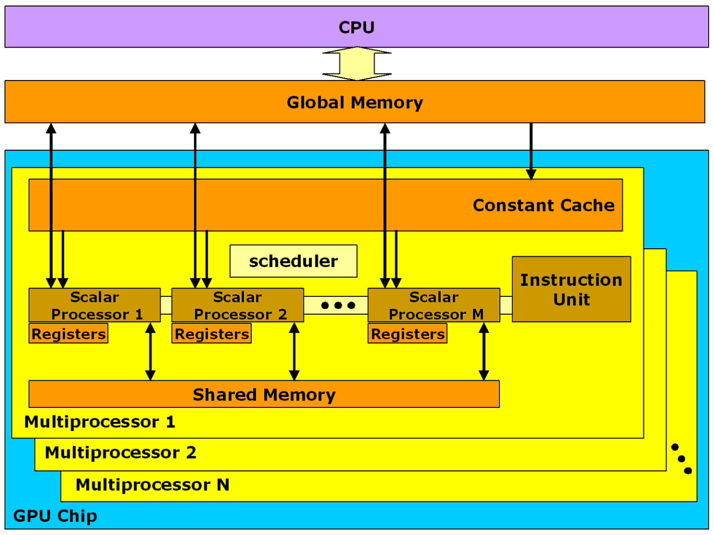

Figure 1: CUDA Architecture.

2.3 Software model

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

In addition, thread assignments in MP are always executed by 32 threads, which are executed simultaneously. This unique definition is called "Warp" in CUDA, as shown in Figure 2. A warp is, so to speak, a common destiny. Threads will dispatch SPs automatically, but it is important to consider the thread unit as 32 for efficient implementation. Both the number of blocks and threads will be defined as larger than the number of SPs sufficient for general CUDA programming because the overhead of the scheduler is very small.

2.4 Memory access system for high memory bandwidth

Although the calculation unit is 32 threads, the memory access unit is 16 threads in CUDA[11].

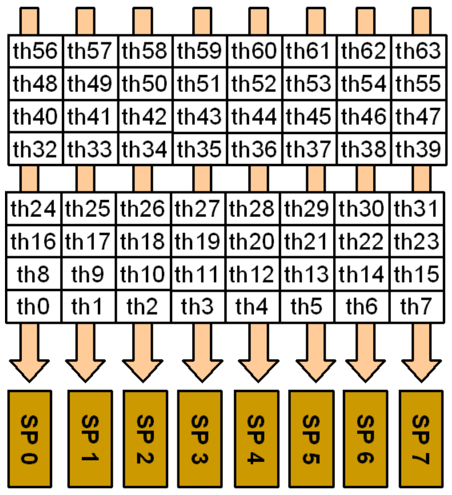
th56 th57 th58 th59 th60 th61 th62 th63
th48 th49 th50 th51 th52 th53 th54 th55
th40 th41 th42 th43 th44 th45 th46 th47
th32 th33 th34 th35 th36 th37 th38 th39
th24 th25 th26 th27 th28 th29 th30 th31
th16 th17 th18 th19 th20 th21 th22 th23
th8 th9 th10 th11 th12 th13 th14 th15
th0 th1 th2 th3 th4 th5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

Figure 2: Warp execution in CUDA.

2.4.1 Coalesced access of global memory

The memory access cycle to global memory takes 400–600 cycles. This latency is fairly low for the access speed in GPU. However, the interface between the global memory and SP is much broader than that with a CPU. To exploit this characteristic, memory access of global memory is hidden by issuing coalesced memory access instruction if the thread would access data with fundamentally no stride access. The CUDA compiler is in charge of whether global memory access becomes coalesced.

2.4.2 Shared memory interleave and the avoidance of bank conflict

Memory access to shared memory has latency as low as that of the registers. Shared memory is divided into 16 banks (4 bytes 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 accesses the same bank, then the memory access will induced serially. At that time, the latency will be 16 times different at maximum depending on how data are allocated to shared memory.

2.5 Techniques for optimization

2.5.1 Pipeline latency hiding

The memory access speed is generally lower than the processing speed because of the read-after-write dependency. According to an experiment by Volcov et al., the SP pipeline latency of one operation between registers takes approximately 24 cycles[13]. In contrast, one warp latency is four cycles. Therefore, the SP pipeline is kept filled by executing at least six warps per MP. Consequently, the latency can be hidden.

2.5.2 Out-of-order execution

With a warp encountering memory stall, another warp will be executed out-of-order to hide this memory stall cycle. Therefore, securing the number of threads to the greatest extent possible hides the latency, thereby improving the overall performance. As shown in section 2.1, we selected an NVIDIA Geforce GTX 285 (Nvidia Corp.) as the GPU, on which up to 32 threads per thread block can be specified.

2.5.3 Cut down of conditional branch

As shown in section 2.1, SPs have no conditional branch unit. Therefore, conditional branch instructions are executed using a pseudo-divergence order generated by the CUDA compiler, not similar to the predication of an Itanium processor (Intel Corp.), but unique in CUDA. First, predication registers are allocated depending on the number of conditional branches to store the result of conditional branch instructions such as an if-or switch. Then, all conditional paths are executed serially, irrespective of whether they are true or false. They are finally decided in relation to the value of the predication registers.

In this way, conditional branching is a main factor of deteriorating processing speed. Therefore reconstruction of the algorithm is necessary to constrain the number of branches.

3 AES

AES, a symmetric block cipher introduced in 2001 by NIST[6], encrypts and decrypts plaintext and ciphertext blocks using a 128-bit, 192-bit, or 256-bit key size. Its calculation unit is 1 byte. This cipher executes the iteration of the same round, for which the number of iterations depends on the key size. For this study, we selected 128-bit key length AES, which consists of 10 rounds. Each round consists of four transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The final round differs slightly from the other rounds: it does not include MixColumns.

As described in this paper, AES is implemented based on optimized ANSI C source code provided as a part of OpenSSL, the open source toolkit for SSL/TLS[12].

Its algorithm defines round processes combined into a transformation simply using a lookup table called "T-box" and exclusive-or operation. Letting a be the round input, which is divided into each 32 bits, the round output e is represented as

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

where T_0, T_1, T_2, and T_3 are lookup tables and k_j is the j-th column of a roundkey. This algorithm includes only four lookup table transformations and four exclusive-or operations.

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

4 Related work

Cook et al. achieved AES encryption at 1.53 Mbps throughput using OpenGL on an NVIDIA Geforce3 Ti200 (Nvidia Corp.)[3]. However, its performance potential is only 2.3% of the CPU (Pentium IV 1.8 GHz; Intel Corp.) performance because the earlier GPU architecture had an insufficient instruction set for general purpose computing. Harrison et al. achieved AES encryption at 870.8 Mbps throughput on a GeForce 7900GT (Nvidia Corp.) using Direct X9 (Microsoft Corp.)[4].

Manavski implemented CUDA-AES achieved an astounding 8.28 Gbps throughput rate with an input size of 8 MB using an NVIDIA GeForce 8800GTX (Nvidia Corp.) [5]. Manavski reports that the thread block size engenders the fastest implementation in CUDA-AES. Its performance improvement can be observed as the number of thread blocks increases. Furthermore, some effort to reduce the usage of the shared memory is reportedly necessary for better performance in CUDA-AES because the shared memory is divided into equally sized memory modules at the time of AES encryption. However, if bank conflict occurs the access must be serialized, which causes slow performance. Nevertheless, no argument has been presented about the concrete allocation of shared memory.

Di Biagio et al. also implemented counter mode AES (AES-CTR) with CUDA using an NVIDIA GeForce 8800GT (Nvidia Corp.) [1]. They achieved a 12.5 Gbps throughput rate with input size of 128 MB considering processing granularity. They defined as fine-grained design a solution exposing the internal parallelism of each AES round. They proposed that four 32–bit word blocks were dispatched to four SPs, with each thread as a fine-grain processing. Moreover, the coarse-grained design was defined as exploiting higher-level parallelism, which exists between independent plaintext blocks. In this granularity, each thread processes each 128-bit plaintext block.

Nishikawa et al. also discussed granularity [8]. They defined 16 bytes/thread when one thread processes one plaintext block (which contains 16 bytes), as was true also for coarse-grained processing suggested by Di Biagio et al. [1]. Other granularities such as 4 bytes/thread and 1 byte/thread were defined in the same manner as 16 bytes/thread. They implemented AES-ECB encoding according to standard AES algorithm without T-box, which achieved 6.25 Gbps with a GeForce GTX285 (Nvidia Corp.). Furthermore, they proposed DES implementation on a GTX285 (Nvidia Corp.), which was intended as a brute force attack[7].

Table 1 shows the respective peak performances and implementation environments of these previous works. This paper presents a discussion of a more detailed implementation of AES referred from these respective previous works, with an attempt to elicit better AES performance using a GTX285 (Nvidia Corp.).

Table 1: Performance of previous methods.

Reference	Device	Language	Throughput
Cook et al.[3]	GeForce3 Ti200	OpenGL	1.53 Mbps
Harrison et al.[4]	GeForce7900GT	DirectX9	870.8 Mbps
Manavski [5]	GeForce8800GTX	CUDA	8.28 Gbps
Di Biagio et al.[1]	GeForce8800GT	CUDA	12.5 Gbps
Nishikawa et al.[8]	GeForce GTX285	CUDA	6.25 Gbps

5 AES encryption implementation on CUDA GPU

This section presents discussion of the granularity of parallel processing and memory allocation to design parallelized AES on a CUDA GPU. Granularity signifies a task size to dispatch to a processor. Granularity is an effect of the parallel AES algorithm design which represents how to parallelize the AES algorithm. The memory allocation strategy of CUDA is important because CUDA has some different types of memory systems, as described earlier. Characteristics of each memory system mutually differ to quite a degree. The difference of memory allocation strategy will provide measurable effects for performance.

5.1 Granularity of parallel processing

Granularities of four types are defined to parallel AES algorithm, as described below.

5.1.1 16 bytes/thread

Using the parallelizing method of 16 bytes/thread means that each thread processes each plaintext block consisting of 16 bytes independently. This implementation presents advantages of lower overhead than other granularities requiring no synchronization and no shared data between threads. This granularity uses a parallelism exploited between plaintext blocks only. Figure 3 shows that threads process plaintext blocks independently.

5.1.2 8 bytes/thread and 4 bytes/thread

Granularity at 8 bytes/thread processes one plaintext block with two threads. Figure 4 presents a plaintext block divided into two threads and two plaintext blocks processed by four processors simultaneously. This method exploits parallelism between plaintext blocks and inner plaintext processing

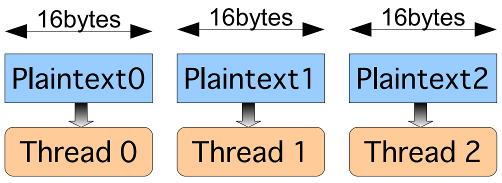

Figure 3: Granularity at 16 bytes/thread.

concurrently. This method requires shared memory to share intermediate data by two threads and synchronization.

Granularity at 4 bytes/thread processes one plaintext block with four threads. Figure 5 portrays this granularity. This method differs from that of 8 bytes/thread in its number of threads for a plaintext block sharing. Shared memory and synchronization are necessary for the same reason as 8 bytes/thread. These granularities exploit more parallelism than 16 bytes/thread, although they require synchronization and shared memory.

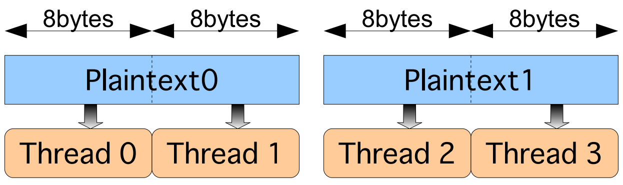

Figure 4: Granularity at 8 bytes/thread.

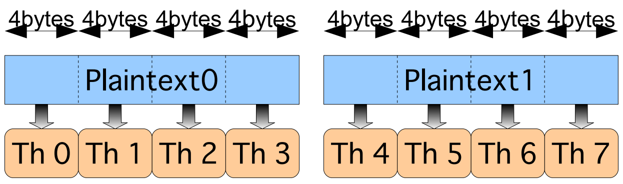

Figure 5: Granularity at 4 bytes/thread.

5.1.3 1byte/thread

It is absolutely better to process AES encoding with at least a 32-bit operating unit because the AES encoding algorithm used in these studies is optimized for 32–bit processing. However, it is able to process 1 byte data by a thread. 1 byte/thread means that 16 threads process a plaintext block in a coordinated manner. This granularity is designed to compare earlier studies and also other granularities, although this granularity will bring unsatisfactory performance for a GPU-equipped 32-bit operation unit.

5.2 Memory allocation

5.2.1 Overview of memory access of parallelized AES with CUDA GPU

AES contains data structures of three kinds as shown as below.

1. Plaintext, ciphertext, and intermediate data. Intermediate data are put in temporary data storage during processing encoding (decoding) of AES.

2. T-box

3. Round key (The round key is calculated from the secret key.)

When encryption is initiated, all data are stored in the main memory of the host computer. At the beginning of AES processing using CUDA GPU, the plaintext, round key, and T-box tables are transferred to global memory and constant memory of GPU. To hasten processing, data are transferred to other higher speed memory such as shared memory and GPU registers. In various memory systems, these data will be transferred. It is necessary to consider the characteristics of these memory systems to store data in their correct place.

To simplify problem modeling, the round key is calculated using the CPU in this paper, although it can be calculated using either a GPU or a CPU. In any case, the calculation cost of the round key is negligibly small when the plaintext is large.

5.2.2 Round key and T-box

T-box and round key are read-only data shared among all threads.

According to such characteristics, these variables match to allocate on constant memory. Constant memory provides very low latency when the cache system works well. With respect to T-box, it requires random access, so the possibility exists that they can not provide low access latency.

Even if a round key is allocated on shared memory, AES will exhibit good performance because shared memory provides low access latency. However, two problems exist. One is a bank conflict, which occurs when threads access the same memory located at the same bank. The other is that allocating round key on shared memory wastes its capacity. Many copied T-boxes will appear because shared memory can be shared between limited threads belonging in the same thread blocks.

5.2.3 Plaintext

The plaintext is stored on the global memory at the beginning of processing. When AES encoding starts, the plaintext will be transferred to shared memory sequentially to share intermediate data between stream processors. For granularity at 16 bytes/thread, intermediate variables calculated from plaintext will be stored on registers directly because this granularity does not share any intermediate variables. Naturally, granularity at 16 bytes/thread can also be used as shared memory instead of registers.

Using shared memory, as shown in Figure 6 and Figure 7, a choice of two options exists about how to allocate data in memory: Array of Structure (AoS) and Structure of Array (SoA). A structure means a plaintext block structure. The AoS deals with plaintext by its nature. The SoA allocates each element of plaintext into one array. They provide a difference of appearance of bank conflict. To reduce bank conflict occurrences, we select a better allocation pattern for each implementation. Figure 6 and Figure 7 also portray examples of the thread access pattern to shared memory for granularity at 8 bytes/thread.

Finally, Table 2 portrays a summary of memory allocation candidates of each datum.

Table 2: Variables of AES and their allocation.

Variables	Constant Memory	Shared Memory	Registers
T-box	○	○	×
Round Key	○	○	×
Plaintext	×	○	○(in case of 16 bytes/thread)

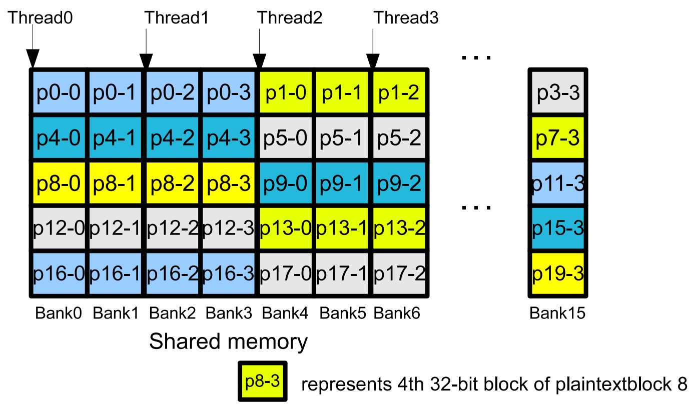
Thread0
Thread1
Thread2
Thread3
p0-0 p0-1 p0-2 p0-3 p1-0 p1-1 p1-2 p3-3
p4-0 p4-1 p4-2 p4-3 p5-0 p5-1 p5-2 p7-3
p8-0 p8-1 p8-2 p8-3 p9-0 p9-1 p9-2 p11-3
p12-0 p12-1 p12-2 p12-3 p13-0 p13-1 p13-2 p15-3
p16-0 p16-1 p16-2 p16-3 p17-0 p17-1 p17-2 p19-3
Bank0 Bank1 Bank2 Bank3 Bank4 Bank5 Bank6 Bank15
Shared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

Figure 6: Array of Structure, an allocation of the plaintext.

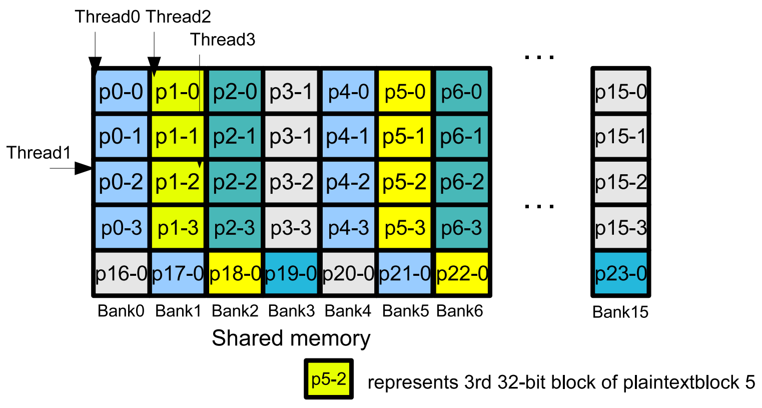
Thread0 Thread2
Thread3
Thread1
p0-0 p1-0 p2-0 p3-1 p4-0 p5-0 p6-0 p15-0
p0-1 p1-1 p2-1 p3-1 p4-1 p5-1 p6-1 p15-1
p0-2 p1-2 p2-2 p3-2 p4-2 p5-2 p6-2 p15-2
p0-3 p1-3 p2-3 p3-3 p4-3 p5-3 p6-3 p15-3
p16-0 p17-0 p18-0 p19-0 p20-0 p21-0 p22-0 p23-0
Bank0 Bank1 Bank2 Bank3 Bank4 Bank5 Bank6 Bank15
Shared memory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

Figure 7: Structure of Array, an allocation of the plaintext.

5.3 Other optimizations

5.3.1 T-box architecture

T-box implementations of several different kinds are evaluated based on well optimized OpenSSL AES implementation. Three different implementations of T-boxes are described here.

1. One 32-bit array T-box and rotate operation. (saving memory area)

2. One 64-bit array precomputed T-box with byte order memory access.

3. Precomputed 32-bit array T-box$\times$4

Actually, these T-boxes are made from same one T-box with rotate and shift operations. In addition, the shift operation can replace byte order memory access. The first one just achieves realization of four T-box operations by one T-box with rotate operations. The byte order access was accepted and provided with high bandwidth. Then the second one would bring in high performance such as x86 architecture. In this paper, implementation of this method is not shown because CUDA GPU is currently not able to operate memory access with byte order.

The last one is optimized implementation in a natural way. Four precomputed 32-bit array T-boxes were implemented.

5.3.2 Cut down of thread block switching

Usually CUDA applications, massively parallel processing data are mapped respectively to each thread. For example in 3D rendering, each pixel or vertex is mapped to each thread. In fluid computation, each particle is mapped to each thread. Similarly in AES, we can map each plaintext to each thread as in the applications above, but the time of one encryption by threads is slightly different. Therefore, the overhead of the switching thread block in AES tends to be larger and not negligible, different from the other applications.

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

5.3.3 Overlapping GPU processing and memory copy

It is necessary to consider overhead attributable to the data transfer between CPU and GPU to exploit effective performance. To hide this overhead, CUDA provides overlapping data transfer (memory copy) and processing. We implemented the AES encoding process with overlapping transferring plaintext to (and ciphertext from) global memory of GPU and GPU's AES process. Figure 8 portrays a diagram of this overlapping in case of data divided into two blocks. Before processing AES on GPU, the transfer of plaintext block one of two to the GPU is begun. When the transfer is finished, AES encoding will be started on GPU and transferring of second plaintext block will be started simultaneously. When the encoding process of the first plaintext block is finished, the encoding process of the second plaintext block will be started because transfer of the second plaintext block to GPU would be finished and the write back process of the first plaintext block to CPU will be started simultaneously. Finally, the write back process of the second plaintext block will be done. It is necessary to synchronize times at which each plaintext block changes its process.

This overlapping process is well known as pipeline processing between data transfer and processing. To optimize this pipeline, a tradeoff exists between block size, which is dividing the size of the plaintext as pipeline stage, and the pipelined overhead. This pipeline optimization will achieve good performance when the data transfer time and processing time are balanced.

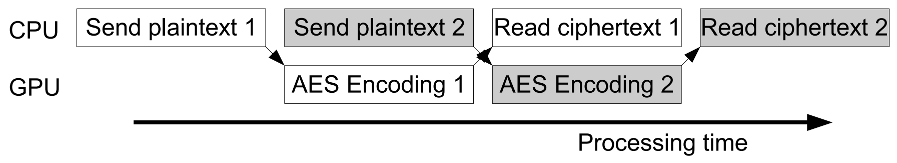

Figure 8: Overlapping data transfer and processing.

6 Implementation results

6.1 Environment

Table 3 shows the specifications of the computer systems used for this experiment.

The implemented AES for this experiment was a 128–bit AES encoding algorithm (ECB mode). The round key was calculated once by the CPU and transferred to GPU's global or constant memory before starting AES processing. Plaintext was generated by CPU with a random value too. The plaintext size was 256 MB fixed size to evaluate around peak performance in each implementation. The number of thread blocks was 60. The number of threads was 512 fixed in all executions.

The number of threads is also an important parameter to optimize programs using CUDA. In this result, the number of threads was fixed at 512 to attain the highest throughput because it was

Table 3: Computer specifications.

CPU	Corei7 Quadi7-920 (2.66 GHz)
Memory	6GB
OS	CentOS5.3 (kernel ver2.6.18)
Compiler	gcc ver4.1.2 (option -O3)
GPU Accelerator	NVIDIA Geforce GTX 285
GPU Memory	1 GB
CUDA Compiler	nvcc ver. 2.3

shown that the difference of the number of threads over 128 produced a limited effect in [1] in case of sufficiently large plaintext size. The number of thread blocks was fixed for the same reason.

6.2 Implementation parameters

According to previous section, AES is implemented with many implementation parameters. Table 4 shows implementation parameters used for this evaluation.

Table 4: Implementation parameters

Granularity	16 B/thread, 8 B/thread, 4 B/thread, 1 B/thread
T-box allocation	Constant Memory, Shared Memory
T-box architecture	Four T-boxes (32 bit), One T-box (64 bit)
Round Key allocation	Constant Memory, Shared Memory
Plaintext allocation	Shared Memory (AoS or SoA), Registers

6.3 Throughput

Tables 5 and 6 show the throughput of each implementation.

Table 5: Throughput, 16 B/thread.

Implementation type	I	II	III	IV	V	VI
T-box allocation	constant	shared	shared	shared	shared	shared
Round key	constant	constant	shared	shared	shared	shared
Plaintext	register	register	register	register	shared	shared
Plaintext Allocation	-	-	-	-	SoA	AoS
T-box architecture	4	4	4	1	4	4
Throughput [Gbps]	5.0	34.4	**35.2**	21.54	29.9	21.7

6.3.1 Throughput with computation granularity

Figure 9 shows the best throughput of each computation granularity. Results show that one out of 16 bytes/thread implementation achieved the highest throughput at 35.2 Gbps. In fact, 16 bytes/thread has constituted an advantage compared with other implementations. They required neither shared memory for processing AES encoding nor synchronization. Requiring no shared memory means bringing in not only high-speed register access but also no memory bank conflict.

Table 6: Throughput, 8B/thread, 4B/thread, and 1B/thread.

Granularity	8 B/th	8 B/th	8 B/th	4 B/th	4 B/th	4 B/th	1 B/th
T-box allocation	shared						
Round key	constant	shared	constant	constant	shared	constant	shared
Plaintext	shared						
Plaintext Allocation	AoS	AoS	SoA	AoS	AoS	SoA	AoS
T-box architecture	4						
Throughput	26.9	23.9	23.4	25.3	25.0	17.1	2.9

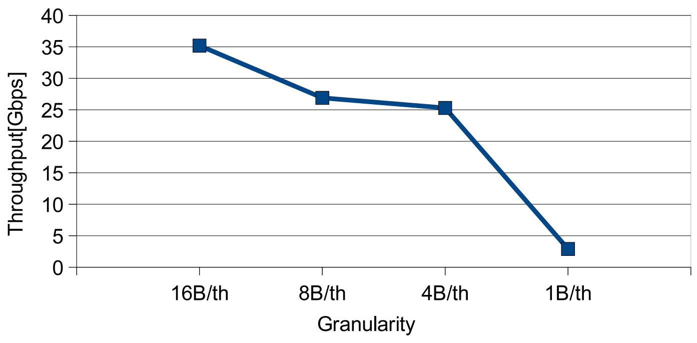

Figure 9: Throughput at each granularity.

Let us next discuss 8 bytes/thread and 4 bytes/thread. The highest throughput was 26.9 Gbps, but they had almost identical throughput. Their throughputs were almost 30% lower than the 16 bytes/thread implementation. The 8 and 4 bytes/threads required shared memory to share plaintexts and intermediate values. But shared memory access brought about bank conflicts. Synchronizations and bank conflicts of shared memory were the major causes of this decline in performance.

Actually, 1 byte/thread achieved very low throughput, but no surprise existed because the implemented algorithm was designed using 8-bit operation made from an algorithm optimized to 32-bit operation divided into four operations.

Finally, the best throughput achieved 29 times higher speed than that of a Core i7-920 2.66 GHz CPU-based implementation, which achieved 1.2 Gbps using OpenSSL optimized source code for the Intel architecture with a single core.

6.3.2 T-box allocation

The difference of performance between constant memory and shared memory must be discussed. Implementation types I and II in Table 5 show throughput in the same cases of T-box architecture, plaintext allocation, and round key allocation except T-box allocation. For implementation type I, it indicated an extremely low throughput at 5.0 Gbps as a result of T-box allocating on constant memory. By contrast, type II implementation achieved a high throughput at 34.4 Gbps as a result of T-box allocating on shared memory. In fact, both constant memory and shared memory have almost equal ability of access latency, but their mechanisms differ. Constant memory achieved low latency by a cache system in this result. It provided very low latency when the regular memory access pattern was required because the cache system works well in such cases. T-box required a random access pattern because it provided nonlinear transformation. As a result, the gap separating implementation types I and II showed a difference of performance of constant memory for random access and regular access. This result revealed that T-box must be implemented on shared memory for AES. It is better for saving a shared memory area to allocate T-box on constant memory, but the shared memory provided sufficient area for T-box in this AES implementation.

Now, T-box architecture is discussed. Implementation type III showed the throughput of four T-box architecture. Type IV showed the throughput of one T-box architecture. Implementation type IV achieved about 40% lower performance than implementation type III. This fact shows that the computation cost of rotation of T-box processing gives an impact for performance. Although four T-box architecture requires memory space, it shows no effect on performance. The difference of T-box architecture produced the same performance gap for all implementation granularities.

Consequently, it is absolutely better to use four T-box architecture. Then the T-box must be allocated shared memory.

6.3.3 Plaintext allocation

Regarding plaintext allocation, except for the 1 byte/thread, both AoS and SoA implementations were evaluated. As described in the preceding subsection, this subsection specifically examines the implementation results which limit the T-box allocation to shared memory and T-box architecture to four.

For computation granularity at 16 bytes/thread, shared memory for plaintext was unnecessary because registers were replaceable by shared memory. In addition, plaintext was able to be stored in the shared memory, but the throughput was 15%–35% lower than the implementation which stored plaintext on registers. Allocating plaintext at registers achieved higher performance than allocating plaintext at shared memory because it elicited no memory access conflict and no memory address computation.

Comparison of the throughput of AoS and SoA implementations reveals that SoA implementation achieved better performance than AoS implementation. According to statistical analyses of bank conflict, SoA implementation at granularity of 16 bytes/thread occurs less than AoS implementation because each plaintext block is placed in each bank.

Granularity at 8 and 4 bytes/thread, plaintext must be stored on shared memory. The difference of performance from AoS and SoA is important. The AoS implementation provided about 50% better performance than SoA implementation at 4 bytes/thread granularity and also 12% better at 8 bytes/thread. Bank conflicts in these granularities are alike according to statistical analysis. Although some cases prospected that SoA implementation encountered less bank conflict than AoS implementation, all results showed that throughputs of AoS implementation were better. It might be affected by the fact that SoA requires more difficult address calculation. This result explains that, in some cases, it is too difficult to analyze bank conflict statistically.

6.3.4 Round key allocation

With respect to allocating the place of the round key, a slight difference was found in performance. Implementation allocating the round key on shared memory was about 2% faster than allocating it on constant memory implementation at granularity 16 bytes/thread. Because round key access necessitates coherent memory access, a cache system must work well. For that reason, they had almost identical throughput.

For Granularity at 4 and 8 bytes/thread, allocating the round key on constant memory led to better performance than allocating shared memory, which result was contrary to that of 16 bytes/thread result. The reason for this difference was bank conflicts, which increased when 4 and 8 bytes/thread used shared memory to store plaintext and intermediate data. Allocation of the round key on constant memory led to lower bank conflicts than those elicited when using shared memory for the round key.

6.3.5 Comparison with previous work

The AES implementation for GPU with consideration of granularity and T-box memory allocation is reported in [1]. They achieved about 12.5 Gbps throughput AES CTR encoding processing for 128 MB size plaintext using an NVIDIA 8800GT (Nvidia Corp.), which has 112 stream processors and a processor clock of 1.5 GHz. The NVIDIA GeForce GTX285 (Nvidia Corp.) used in these experiments has 240 stream processors; its processor clock is about 1.5 GHz. Comparing them, two times or even greater performance gaps exist not only because of the number of processors, but also the architecture of the processor and CUDA runtime libraries differ. Actually, they were three times faster at the maximum throughput, as explained in this paper than [1]. The highest throughput in [1] was achieved with four T-box architecture, T-box allocated on shared memory and 16 bytes/thread granularity. These implementation circumstances are the same as the highest throughput achievement setting presented in this paper. This paper also presented new factors such as more granularity and varied memory allocation patterns. Some contributed to better performance.

6.4 Overlapping data transfer

The performance described so far in this paper has excepted data transfer time for discussion of the effectiveness about difference of granularity, memory allocation and others. To evaluate the real effectiveness for performance provided by GPU, data transfer overheads between the CPU and GPU must be considered.

Implementation results were evaluated with AES plaintext size at 256 MB. As inferred from measurements of data transfer times, there is about 0.08 ms overhead, which includes both copies to and from global memory from and to the CPU. Throughput including these data transfer times at 16 bytes/thread (four T-boxes and round key allocated on shared memory) achieved only 13.4 Gbps. We implemented another AES encoding program, which is applied to overlapping data transfer and AES processing to exploit better performance. Results show that implementation applied overlapping to 16 bytes/thread (four T-boxes and round key allocated on shared memory) achieves 22.5 Gbps, meaning that overlapping can hide 65% of the data transfer time.

This implementation divided plaintext data into four plaintext blocks. Also, AES encoding programs worked four times independently. Based on this experience, although dividing plaintext into four blocks achieves the best performance, more investigations must be made to elucidate overlapping data transfer and processing.

7 Conclusion

This paper presented an analysis of the effectiveness for AES implementation from various conditions such as parallel processing granularity and various memory allocations. Greater than 10 times difference of performance and the best AES implementation were achieved: 35.2 Gbps throughput using GPU.

Results showed that such implementation granularity at 16 bytes/thread tends to be effective. Moreover, the common data table T-box and round key allocated on shared memory achieved the best performance, but round key, which requires coherent access, will be able to allocate constant memory. Results show that AES must implement granularity at 16 bytes/thread, with four T-box architecture and allocation of T-box on shared memory for efficient processing with the GPU. However, round key allocation place is ineffective for performance whether a round key is allocated on shared memory or constant memory.

The best result achieved in this study was 35.2 Gbps AES encoding throughput, which strongly suggests the great potential of the CUDA GPU for use as a cryptographic accelerator. Regarding high-end applications, the possibility exists for application of a code breaker for example, but we also expect a small accelerator for a block cipher accelerator because most computers such as notebook PCs are also equipped with a GPU.

Moreover, these analyses revealed an overlap between data transfer and GPU processing, which supported later versions of CUDA, achieving 22.5 Gbps throughput including data transfer overhead. This throughput implies 68% faster processing compared to that without overlapping. Moreover, overlapping hid 65% of the data transfer time, which showed that the GPU becomes a real accelerator for AES encoding with data transfer overlapping.

We are developing an automatic parallelization technique for cipher algorithms including AES on GPU. These results are the first step toward automatic decision for memory allocation and computation granularity. However, a long road of development separates us from our goal. Consequently, using results described in this paper, we are implementing other block cipher algorithms such as MISTY for GPUs.

References

[1] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. *Parallel and Distributed Processing Symposium, International*, pages 1–8, 2009.

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[3] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. Cryptographics: Secret key cryptography using graphics cards. In *In RSA Conference, Cryptographerfs Track (CT-RSA)*, pages 334–350, 2005.

[4] Owen Harrison and John Waldron. AES encryption implementation and analysis on commodity graphics processing units. In *9th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 209–226, 2007.

[5] Svetlin A. Manavski. CUDA compatible gpu as an efficient hardware accelerator for AES cryptography. In *IEEE International Conference on Signal Processing and Communication, ICSPC*, pages 65–66, 2007.

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

[7] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. Acceleration of the key crack against cipher algorithm using CUDA (in Japanese). In *IEICE technical report. Computer systems 109 (168)*, pages 49–54, Sendai, Japan, July 2009.

[8] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. Granularity optimization method for AES encryption implementation on CUDA (in Japanese). In *IEICE technical report. VLSI Design Technologies (VLD2009-69)*, pages 107–112, Kanagawa, Japan, January 2010.

[9] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html, 2009.

[10] NVIDIA CUDAZONE. http://www.nvidia.com/object/cuda_home.html, 2009.

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

[12] The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org, 2007.

[13] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.