PAPER

High Throughput Parallelization of AES-CTR Algorithm

Nhat-Phuong TRAN[†], *Nonmember*, Myungho LEE[†a)], *Member*, Sugwon HONG[†], *and* Seung-Jae LEE[††], *Nonmembers*

SUMMARY Data encryption and decryption are common operations in network-based application programs that must offer security. In order to keep pace with the high data input rate of network-based applications such as the multimedia data streaming, real-time processing of the data encryption/decryption is crucial. In this paper, we propose a new parallelization approach to improve the throughput performance for the de-facto standard data encryption and decryption algorithm, AES-CTR (Counter mode of AES). The new approach extends the size of the block encrypted at one time across the unit block boundaries, thus effectively encrypting multiple unit blocks at the same time. This reduces the associated parallelization overheads such as the number of procedure calls, the scheduling and the synchronizations compared with previous approaches. Therefore, this leads to significant throughput performance improvements on a computing platform with a general-purpose multi-core processor and a Graphic Processing Unit (GPU).

key words: *AES, multi-core, GPU, parallelization*

1. Introduction

Recently, with the widespread use of the Internet in commercial applications, network-based programs are becoming increasingly popular. In order to protect the copyright to the contents of such applications, data encryption and decryption are essential. Among many encryption/decryption standards, Advanced Encryption Standard (AES) is a representative one. AES is a symmetric cryptographic algorithm published by NIST [4] and is widely used recently because of its high security and the low cost. AES algorithm is carried out by applying a number of transformation rounds that convert the input plain text into the final cipher text. The output of each round is fed back to the next round as the input. Each round includes several computation steps such as XOR operations, byte substitutions, shift rows, mix columns which require matrix computation operations and table lookups. In each round, a new key is generated and used for the above computation steps. For the encryption and decryption of multiple blocks, AES has several modes of operations [6]. In this paper, we use the Counter (CTR) mode of AES which is parallel in nature and secure by using different keys in blocks.

Manuscript received May 11, 2012.
Manuscript revised February 17, 2013.

[†]The authors are with the Dept. of Computer Science and Engineering, Myongji University, 38–2 San Namdong, Cheo-In Gu Yong In, Kyung Ki Do, 449–728 Korea.
[††]The author is with the Dept. of Electrical Engineering, Myongji University, Korea.
a) E-mail: myunghol@mju.ac.kr (Corresponding author)

DOI: 10.1587/transinf.E96.D.1685

Since mid-2000, incorporating multiple CPU cores on a single chip (or multi-core processor) has become a main stream microprocessor design trend. As a Chip Multi-Processor (CMP), a multi-core processor can execute multiple software threads on a single chip at the same time. Thus it can provide higher computing power per chip for a given time interval (or throughput) [19]. The multi-core design trend has also appeared in the recent Graphic Processing Unit (GPU) by incorporating Shader, Vertex, Pixel units—separate processing units in the earlier GPUs—into uniform programmable processing units or cores [15]. These processing units or cores can be programmed and executed in parallel. This architectural innovation led to the excellent floating-point performance (flops) for the GPU. In addition to the architectural changes, user friendly parallel programming environments have been recently developed (e.g., Nvidia's CUDA, Khronos Group's OpenCL) which provide programmers with more direct control of the GPU pipeline and the memory hierarchy. Using these environments along with the flexible multi-core GPU architecture has led to innovative performance improvements in many application areas besides the graphics and many more are still to come [15].

In a network-based application program that must offer security, the data is received continuously with a high input rate. In order to keep pace with the high rate of the data input, a real-time processing of the data encryption/decryption is crucial. In this paper, we develop a new parallelization technique to improve the throughput performance of the standard encryption/decryption algorithm, AES-CTR, which needs real-time processing. The new approach parallelizes the AES-CTR by extending the block size across the unit block boundaries where the data encryption is originally applied. Thus this approach effectively encrypts multiple unit blocks at the same time. This reduces the associated parallelization overheads such as the number of procedure calls and the job scheduling, and the synchronizations compared with the previous approaches. By implementing the proposed approach on a computing platform with a general-purpose multi-core processor (2.2 Ghz 4-core Intel processor) and a GPU (Nvidia GeForce 8800 GT), we've observed significant throughput performance improvements compared with previous parallelization approaches. In fact, our approach leads to the 7.25-times speedup and the higher throughput performance compared with the previous coarse-grain parallelization approach. The resulting throughput performance reaches up to 87 Gbps on Nvidia GeForce 8800 GT GPU.

The rest of the paper is organized as follows: Section 2 gives an overview of the AES algorithm and its modes of operations including the CTR mode. Section 3 shows the architecture of the latest general-purpose multi-core processors and multi-core GPUs, and their programming models. Section 4 describes previous research employing the fine-grain and the coarse-grain parallelization. Section 5 explains our approach compared with the previous approaches. Section 6 shows the results of experiments of the new approach compared with the previous approach on a 4-core Intel processor and Nvidia GeForce 8800 GT GPU. Section 7 wraps up the paper with conclusions.

2. Overview of AES Algorithm

The Advanced Encryption Standard (AES) is a symmetric cryptographic algorithm published by NIST [13] which had replaced the previous Data Encryption Standard (DES). AES is the most widely used block cipher algorithm in recent years because of its high security and low cost. In order to encrypt and decrypt a sequence of blocks, modes of operation have been developed. In this section, we describe the main computation steps for AES block cipher algorithm first and then describe its modes of operation.

2.1 Computation Steps

AES algorithm is carried out by applying a number of repetitions of transformation rounds that convert the input plain text into the final cipher text. AES has a fixed block size of 128 bits with three key lengths of 128 bits, 192 bits, and 256 bits, thus comprises three block ciphers AES-128, AES-192, and AES-256. Depending on the key and the block lengths, the number of rounds of AES is varied: 10 for 128 bits, 12 for 192 bits, and 14 for 256 bits. Each round includes several steps. The output of each round is fed back as the input of the next round. Each round consists of the same steps, except for the first round where an extra addition of a round key is performed and for the last round where the step for the mixing columns is skipped [3], [4], [6].

Figure 1 below shows the steps of the AES-128 algorithm which is iterative with 10 rounds. The input to the algorithm is a block of 128 bits plain text which is represented by a 4×4 byte matrix called "State". The operations performed for each step are as follows:

- *KeyExpansion* is used to generate the *RoundKeys* from the original key for rounds.
- The four round steps are *AddRoundKey* (XOR each column of the State with a word from the key schedule), *SubBytes* (process the State with non-linear byte substitution table, S-box, that operates on each of the State bytes independently.), *ShiftRows* (cyclically shifts the last three rows in the State by different offsets), *MixColumns* (takes all of the columns of the State and mixes their data to produce new columns).
- Operations performed in each round are as follows:
  - In the initial round, perform the *AddRoundKey* operation and the *SubBytes*, *ShiftRows*, *MixColumns,* and *AddRoundKey*. Thus the *AddRoundKey* operation is performed an extra time.
  - In the next *N*–1 rounds, perform four operations *SubBytes*, *ShiftRows*, *MixColumns,* and *AddRoundKey*.
  - In the last round, perform the same operations of the previous *N*–1 rounds except the *MixColumns* operation.

  More detailed description on the above operations can be found in [3].

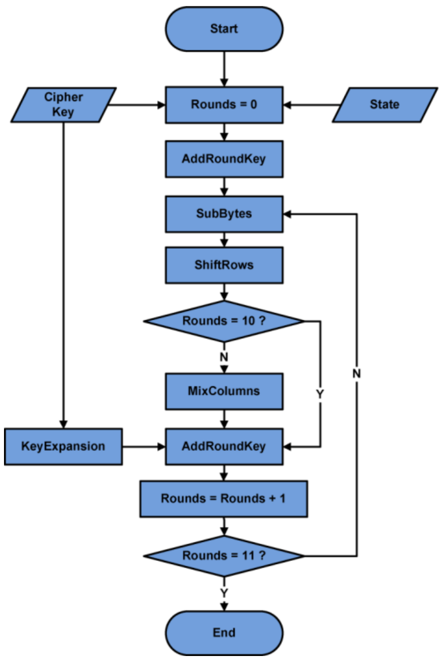


Fig. 1 Computation steps for AES-128 algorithm.

Besides the matrix computation operations used for *AddRoundKey, SubBytes, ShiftRows*, and *MixColumns* applied in each round, the table lookup is another important operation. With the table lookup, the different steps of the round can be combined in a single set of table lookups as the following formula shows:

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

where $a_{i,j}$ refers to the input matrix variable, e_j refers to the output matrix in each round transformation, k_j is the j-th word of the expanded key. T_0, T_1, T_2, and T_3 refer to the lookup tables which have 256 32-bit word entries each and are made up of 4 KB of the storage space and obtained through combinations as follows:

$$T_0[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \end{bmatrix} \quad T_1[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \end{bmatrix}$$

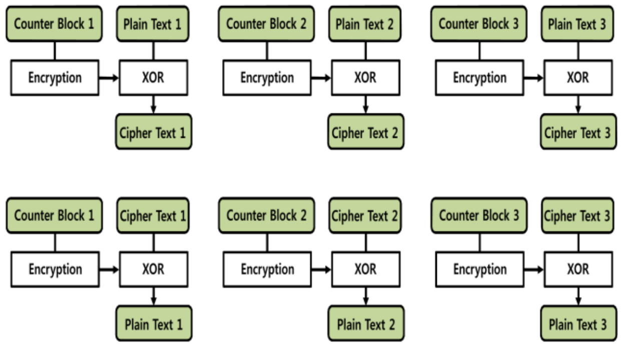


Fig. 2 Example applications of AES-CTR algorithm.

$$T_2[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \end{bmatrix} \quad T_4[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \end{bmatrix}$$

where $\bullet$ is a GF(2^8) finite field multiplication [3].

2.2 Modes of Operation

In order to encrypt and decrypt a sequence of blocks in the AES, a number of modes of operation have been developed [6]. In the Electronic CodeBook (ECB) mode, a given plain text is divided into multiple unit-sized blocks, which are then encrypted or decrypted independently. Thus this mode is parallel in nature, but not secure because the same plain text can be encrypted into the same cipher text. In the Cipher Block Chaining (CBC) mode, in order to encrypt a unit-sized block k, the cipher text for the previous block $k-1$ is used. Thus it is working in a chained fashion and cannot be parallelized. Cipher FeedBack (CFB) mode and Output FeedBack (OFB) mode are close relatives of CBC mode. Thus these modes cannot be parallelized either.

In this paper, we use Counter (CTR) mode because it is parallel in nature and secure by using different keys in blocks. In this mode, we denote the length of plain text blocks to be m. A *keystream*, denoted by z_i, is produced by choosing *counters*, denoted by *cter*, whose length is also m bits. Then we produce counter T_i by $T_i = (cter + i - 1) \bmod 2^m$. Then encrypt the plain text blocks by $c_i = p_i \oplus E_k(T_i)$. Figure 2 shows an example encryption and decryption of AES-CTR algorithm.

3. Overview of Architectures

In this section, we first describe the architecture of a general-purpose multi-core microprocessor. Then we describe the architecture of a Graphic Processing Unit (GPU) and the programming model we use for our experiments in the paper.

3.1 Multi-Core Processor Architecture

Recently, microprocessor designers have been considering many design choices to efficiently utilize the ever increasing effective silicon area with the increase of transistor densities. Instead of employing a complicated processor pipeline on a chip with an emphasis on improving single software thread's performance, incorporating multiple processor cores on a single chip (or multi-core processor) has become a main stream microprocessor design trend. As a Chip Multi-Processor (CMP), it can execute multiple software threads on a single chip at the same time. Thus a multi-core processor provides a larger capacity of computations performed per chip for a given time interval (or throughput) [19]. All of the CPU vendors including Intel, AMD, IBM, Oracle/Sun, among others have introduced the multi-core processors to the market. The multi-core design is also adopted in embedded systems such as ARM11 MPcore (quad-core) processor based systems introduced lately.

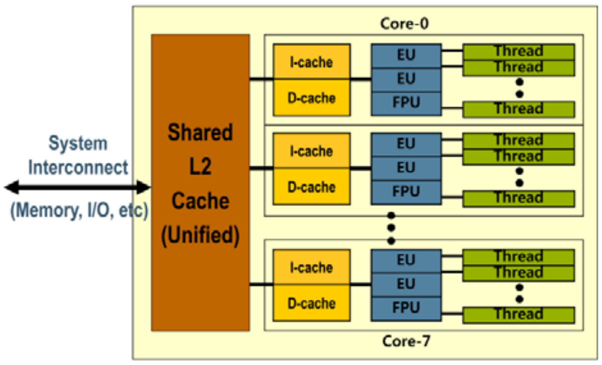


Fig. 3 Architecture of an advanced multi-core processor.

In addition to the CMP based multi-core design, some recent designs go one step further to incorporate the Simultaneous MultiThreading (SMT) or similar technologies such as the Hyper-Threading on a processor core to increase the on-chip thread-level parallelism. Examples are Intel Nehalem and Oracle/Sun UltraSPARC T2/T3 microprocessor. Figure 3 shows the architecture of an advanced multi-core processor. On each processor chip, there are N-processor cores, with each core having its own level-1 on-chip cache. The N-cores share a larger level-2 cache on the processor chip. Each core also has M hardware threads performing SMT or similar functions. Thus it supports two levels of parallelism. For example, the UltraSPARC T2 from Sun includes 8 cores on a chip, with each core supporting 8 hardware threads. In total, 64 ($= 8 \times 8$) threads can execute on a chip at the same time. Each core has 8 KB private data cache. The level-2 unified cache is 4 MB in size.

Although multi-core processors promise to deliver higher chip-level throughput performance than the traditional single-core processors, it is not quite straightforward to exploit its full performance potential. Resources on the multi-core processors such as cache(s), cache/memory bus, functional units, etc., are shared among the cores/threads on the same chip. Software processes or threads running on the cores/threads of the same processor chip compete for the shared resources, which can cause conflicts and hurt performance. Thus efficiently utilizing the multi-core processors is a challenging task [19].

3.2 GPU Architecture and Programming

The Graphic Processing Unit (GPU) was introduced in the late 1990s as a co-processor for accelerating the simulation and visualization of 3D images commonly used in applications such as game programs. Since then the GPU has become widespread and these days it is commonly incorporated in many computing platforms including desktop PCs, high performance computing servers, and even in mobile devices such as smart phones.

In the latest GPU, the clock rate has ramped up significantly compared with the earlier GPUs. Furthermore, the processing units for Shader, Vertex, Pixel which were designed as separate processing units in the earlier GPUs are incorporated into multiple uniform programmable processing units or thread processors [14]. Thus, the recent GPU architecture reflects the multi-core design appearing in the multi-core microprocessors. It is suitable for SIMD (Single Instruction Multiple Data) processing by having multiple threads assigned to each thread block executing the same instructions managed by the Instruction Unit with respect to different portions of data streaming from the global memory to the on-chip memories (shared memory, registers, etc.) on the same thread block (see Fig. 4). The increase in the clock rate and the new design made possible the impressive floating-point performance of the GPU in flops, far exceeding that of the latest CPUs.

In order to utilize the advanced flexible architecture of the GPU, more user friendly programming environments have been recently developed. CUDA from Nvidia and OpenCL from Khronos Group are good examples of such software environments [15], [16]. Using those environments, programmers can have more direct control over the GPU pipeline and the memory hierarchy. The flexible GPU architecture and the user friendly software development environments have led to a number of innovative performance improvements in many applications and many more improvements are still to come [15], [20].

In the experiments conducted in the paper, we use Nvidia's GPU and CUDA. For executing CUDA programs, a hierarchy of memories is used on the Nvidia's GPU. They are registers and local memories belonging to each thread, a shared memory used in a thread block and shared by threads belonging to the block, and the global memory accessed from all the thread blocks [15], [16]:

- Global memory is an area in the off-chip device memory. (The typical size of the device memory ranges from 256 MB to 6 GB. In the GPU that we use for our experiments, Nvidia 8800 GT, the device memory is 512 MB in size.) Through the global memory GPU can communicate with the host CPU.
- Shared memory sits within each thread block and shared amongst the threads running on the multiple thread processors. The management of the shared memory is under the programmer's control. The typical size of the shared memory is 16 KB. The access time closely matches with the register access time, thus it is a very fast memory.
- On a high-end Nvidia GPU such as the Tesla, there is a level-1 (L1) data cache per each thread block. Unlike the shared memory, L1 data cache is a hardware-managed cache. The typical size of the L1 data cache is 48 KB. Or the user can freely set the size of L1 data cache and the shared memory out of 64 KB combined total size of the on-chip memory embedded on a thread block. In Nvidia 8800 GT, there is no L1 data cache. Thus we use the shared-memory only as the fast on-chip memory.
- Registers are used for temporarily storing the data used for computations for each thread, similar to CPU registers.
- Also each thread has its own local memory area in the device memory to load and store the data needed for the computations. For example, when the registers spill during the computations. Since the local memory is an area in the device memory, it is also a slow memory.
- Besides the above memories, there are constant memory and texture memory in the device memory. Data in constant/texture memory are read-only. They can be cached in the on-chip constant cache and the texture cache respectively.

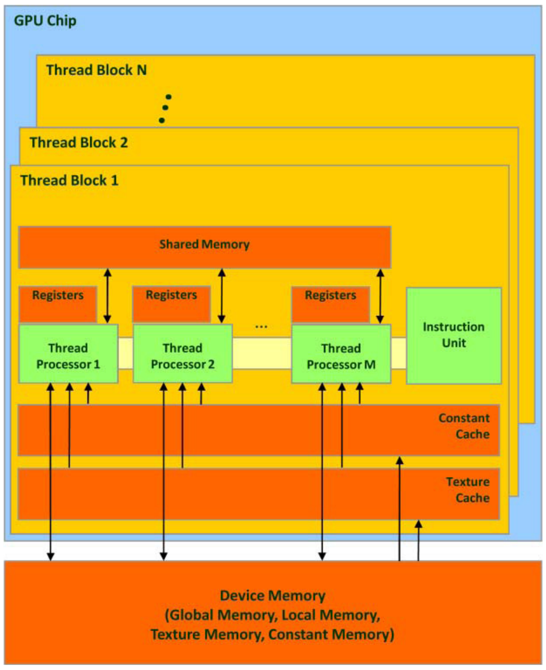


Fig. 4 General architecture of a GPU [15].

In CUDA programs, data needed for computations on the GPU is transferred from the host memory to the global memory, optionally placed in the shared memory by the programmer, and used by thread blocks and thread processors through the registers. The multiple threads assigned to each thread block executes in the SIMD mode by having the same instruction managed by the Instruction Unit on different portions of data as explained earlier in this section. When a running thread encounters a cache miss, for example, the context is switched to a new thread while the cache miss is serviced for the next 200 hundred cycles or more. Thus the GPU is executing in a multithreaded fashion.

4. Previous Research

In a typical network-based application that must offer security, data encryption and decryption are intensively performed with respect to the large amounts of data continuously received from and forwarded to other senders/receivers. In such an environment, the demand for parallel execution of the AES is high. There have been a number of previous attempts to parallelize the AES. We describe them below.

J.W. Bo, et al. [2] presented a software speed record for both the encryption and the decryption using AES on 8-bit microcontroller, Nvidia GPUs, and the Cell Broadband Engine. Harrison and Waldron [9] proposed a study of AES implementation on the GPU hardware, using Nvidia GeForce 6 and 7 series. This implementation is based on the OpenGL library which is not geared towards a general purpose computing. In [1], Harrison and Waldron also presented another implementation of AES with an application oriented approach on GPUs. In their implementation on Nvidia G80, they achieved 4∼10 times speedup over a CPU implementation. Manacski [12] implemented CUDA-AES which runs up to 20 times faster than the implementation of OpenSSL on a general-purpose CPU.

Besides the above previous work, there also has been a previous research on parallelization for the CTR mode of AES (AES-CTR). Andrea D. Biagio, et al. [1] proposed a coarse-grain parallelization approach and a fine-grain approach for the AES-CTR on Nvidia GeForce 8400 GS and 8800 GT GPUs using CUDA. They used the shared memory and the constant memory alternatively to store the lookup tables. In order to maximize the performance of the shared memory approach, they carefully arranged the data placement to avoid the bank conflicts.

Since our parallelization approach enhances upon the coarse-grain and the fine-grain approaches used in [1], we describe those approaches in detail. As explained in Sect. 2, the data encryption in AES-CTR goes through a number of computation rounds with respect to a unit sized block. Within each round, four computation steps involving XORs, byte substitutions, shift rows, and mix columns are performed with respect to each block. Figure 5 shows the two main computation routines in the AES-CTR:

- The *encrypt_block* function is used to encrypt one 16-byte unit block. This function consists of 3 steps: 1) create a new key from an initial key and an initial vector; 2) execute a number of rounds to encrypt a 16-byte data block consisting of *AddRoundKey, SubBytes, ShiftRows, MixColumn*; 3) copy the encryption result back to a result array.
- The *aes_ctr* function contains a for-loop which is used to call the *encrypt_block* function a number of times until all the given 16-byte blocks are encrypted. For example, given 1 KB of data, *aes_ctr* function calls *encrypt_block* function 64 (= 1024/16) times.

```
function aes_ctr ()
begin
    for each 16-byte unit block in the given data
           call encrypt_block ()to encrypt each
           16-byte unit block
    end for
end

function encrypt_block ()
begin
    1.   create a new key from an initial key and
         an initial vector

    for(rounds = 1; rounds <= r; rounds++)
    2.   execute computations of encryption:
         AddRoundKey, SubBytes, ShiftRows, and
         MixColumn
    end for

    3.   copy result back

end
```

Fig. 5 Main routines in AES-CTR using 16-byte unit block size.

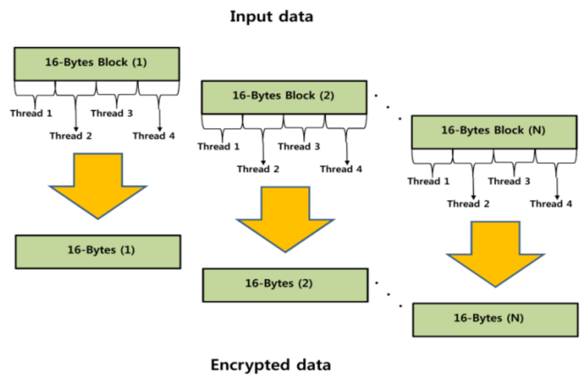


Fig. 6 Fine-grain parallel encryption of 16-byte blocks using 4-threads.

The fine-grain parallelization approach parallelizes the computation steps 1, 2, and 3 in *encrypt_block* function in Fig. 5. Thus each computation step is divided into multiple chunks and assigned to multiple threads for the parallel execution (see Fig. 6). The coarse-grain parallelization approach attempts to parallelize the AES-CTR algorithm at the level of 16-byte blocks. A large amount of data is typically received at a computing node in a network-based application. The data consists of multiple 16-byte blocks. The data encryption is applied to the multiple blocks using multiple threads at the same time [1]. In order to implement the coarse-grain parallelization, the for-loop in the procedure *aes_ctr* in Fig. 5 is parallelized. Figure 7 illustrates the coarse-grain parallelization approach. For example, Block-1, Block-($N/4 + 1$), Block-($N/4 \times 2 + 1$), and Block-($N/4 \times 3 + 1$) are encrypted at the same time by 4 different threads. Then move on to the next 4 blocks (Block-2, Block-($N/4+2$), Block-($N/4\times2+2$), and Block-($N/4\times3+2$)), and so on.

5. Our Parallelization Approach Using Extended Block Size

The fine-grain approach of the previous research parallelizes the encryption of each 16-byte unit size block. The time to execute computation steps is relatively small compared with the time for forking and joining threads. Therefore, it incurs large synchronization overheads. On the other hand, the coarse-grain approach in the previous research does not attempt to parallelize the encryption of single block. Instead, it attempts to parallelize and speed-up the encryption of the total number of blocks in the given data using multiple threads. This approach may lead to a longer run time to encrypt a single block, but incurs significantly lower parallelization overheads such as synchronizations compared with the fine-grain approach. Furthermore, the scheduling overhead is lower in the coarse-grain approach as the number of the parallel task invocations is reduced. In fact, the coarse-grain approach gives better throughput performance overall than the fine-grain approach in encrypting multiple blocks of the given data [1]. In our new parallelization approach, we improve the previous coarse-grain approach by further reducing the overheads associated with the parallelization.

First, we analyze the performance of the previous coarse-grain approach. Given a data consisting of N 16-byte blocks for encryption (for example, if the given data size is 1 KB, then there are $N = 1024/16 = 64$ blocks), the coarse-grain approach distributes N/P-blocks to each core where P is the number of cores available for the parallel execution. Each core encrypts the assigned 16-byte blocks sequentially N/P-times as the Figure 7 shows. (In Fig. 7, P is assumed to be 4.) Each of the N/P repetitions also involves procedure calls for encrypting a block, and the parallel job scheduling and the synchronization overheads. Thus, the parallel time to encrypt N 16-byte blocks can be formulated as

$$T_{parallel} = \frac{N}{P} \times T_{comp} + \frac{N}{P} \times (T_{sync} + T_{sched} + T_{ovhd})$$

Comparing the computation time and the parallelization overhead in the above formula the former is relatively small, because the unit block size (16-bytes) assigned to each core for encryption at one time is small compared with the computing capability of each CPU core.

In order to exploit the computing power of each CPU core more efficiently, we need to increase the granularity of the computation involved in the data encryption so that the associated parallelization overheads can be reduced. To this

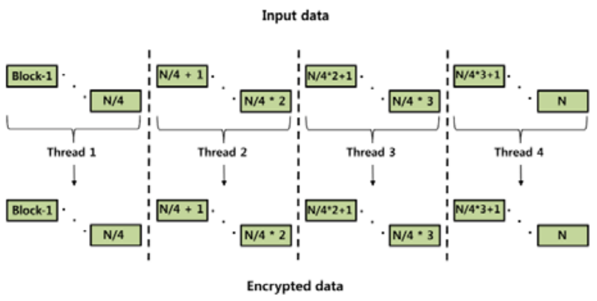


Fig. 7 Coarse-grain parallel encryption of 16-byte blocks using 4-threads.

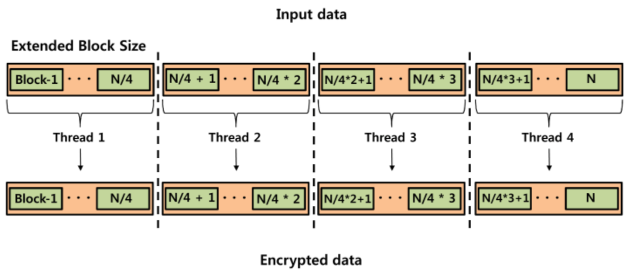


Fig. 8 New approach to parallelize AES-CTR using an extended block.

end, we propose to extend the block size across the 16-bytes unit block boundaries to create a larger block. For instance, we coalesce E-unit blocks (in Fig. 8, $E = N/4$) to create a larger extended block.

Now, we analyze the performance of the proposed approach. Let the computation time for encrypting an extended block ($E \times$ 16-byte) be T_{comp_new}. The computing time for the new approach can be computed as $(N/E)/P \times T_{comp\_new}$. In the new approach, the cost for distributing blocks to each core and the number of procedure calls for the encryption and the synchronizations at the end decreases by a factor of E compared with the coarse-grain approach for a given data size. The formula below summarizes the time for the new approach:

$$\begin{aligned} T_{parallel} &= \frac{N/E}{P} \times T_{comp\_new} \\ &\quad + \frac{N/E}{P} \times (T_{sync} + T_{sched} + T_{ovhd}) \end{aligned}$$

Comparing T_{comp_new} and T_{comp}, we may assume, without loss of generality, that $T_{comp\_new} \leq E \times T_{comp}$, thus $N/E/P \times T_{comp\_new} \leq N/P \times T_{comp}$ by multiplying both sides with $N/E/P$. As mention above, the parallelization overhead decreases by a factor of E (the degree of block coalescing). Therefore, our proposed approach improves the total time to encrypt a given data consisting of N 16-byte blocks. Figure 9 describes the new approach in the pseudo-code.

The new approach incurs some overheads also:

- In order to apply the same key to the extended block, we need to extend the key size also. Thus we first allocate $E \times$ 16-byte memory for the extended key. Then we replicate the 16-byte (size of the unit block) key E-

```
function aes_ctr_new_approach()
begin
   // parallelize the following for-loop
    for each extended block in the given data
       call encrypt_block_new_approach() to
       encrypt each extended block
    end for
end

function encrypt_extended_block()
begin
    create extended key from an initial key and
    an initial vector

    for(rounds = 1; rounds <= r; rounds++)
        execute computations of encryption for
        each extended block corresponding to
        each round

        store temporary results of extended
        block for each round

    end for

    copy encrypted result back

end
```

Fig. 9 Parallelization of AES-CTR using an extended block.

times to fill the allocated memory to be used for the steps afterwards.

- Also, this approach increases the time to encrypt unit-sized (16-byte) block, since the encryption is now performed at the level of the extended block (E×16-bytes). Thus it improves the throughput performance at the cost of the increased latency for encrypting a unit-sized block, because the unit block is now encrypted as part of encrypting E-unit blocks.

6. Experimental Results

We've conducted experiments to measure the performance of our new parallel approach using the extended block. We also measure the performance of the previous parallelization approaches to compare with our approach. The experiments were conducted on both a general-purpose multi-core processor and on a GPU. We present the results in the following subsections.

6.1 Results on General-Purpose Multi-Core Processor

We've parallelized the AES-CTR code for both the coarse-grain approach of the previous research and our new approach using OpenMP [17]. Experiments were conducted on 2.2 Ghz, 4-core Intel Core 2 Duo processor with 2 GB DRAM, running Centos 5.5 OS.

We've also implemented the fine-grain parallelization approach of the previous research. However, it generated a large number of false-sharing of the cache blocks, because multiple threads participate in encrypting single 16-byte block leads to multiple concurrent accesses to the same 16-bytes in the same cache block (at least one of the accesses is a write access). The false-sharing led to prohibitively large run times, at least 10-times slower than the serial execution time. Thus we are not showing the results for the fine-grain approach here.

Table 1 Performance results in seconds on 4-core Intel processor.

Data Size (MB)	Coarse-grain approach			New approach with extended block size								
	No of threads			1KB Block			2KB		4KB		512-byte	
	1	4	8	1	4	8	4	8	4	8	4	8
4	.248	.063	.065	.200	.051	.049	.061	.082	.064	.085	.060	.060
16	1.04	.263	.254	.801	.212	.178	.250	.218	.248	.250	.230	.220
32	1.99	.505	.508	1.56	.395	.400	.480	.450	.482	.460	.458	.456

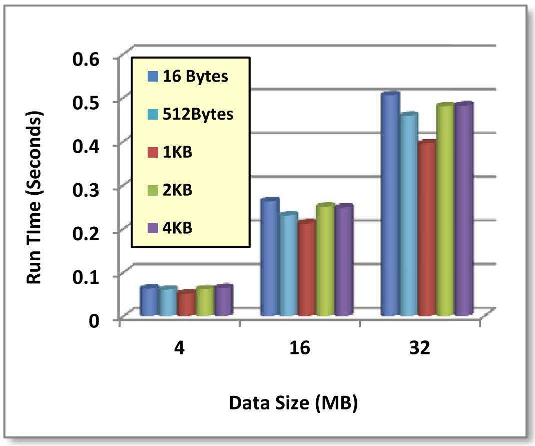


Fig. 10 Run time comparisons of the coarse-grain approach and the new approach using 4-threads and various extended block sizes.

Table 1 compares the run times of the coarse-grain approach and the new approach, using 1-,4-,8-threads on 4 CPU cores. Thus 4-, 8-threads runs used all of the 4 CPU cores.

Figure 10 compares the run times of the coarse-grain approach (16-bytes) and the new approach (1 KB, 2 KB, 4 KB, 512 B) using 4-threads:

- Both the coarse-grain approach and the new approach show good scalability when comparing the run times using 1-thread and 4-threads.
- Using 1 KB extended block size, extended from 16-byte by 64-times, the new approach shows 1.25~1.28x speedup compared with the coarse-grain approach using 4-threads as Fig. 10 shows. The performance improvements are almost uniform for different data sizes (4 MB, 16 MB, 32 MB).
- 1 KB block size turns out to be the best. 512-bytes,

2 KB and 4 KB block sizes also show some improvements for large data sizes such as 16 MB and 32 MB. This is somewhat out of our expectation, because it is speculated that the larger the block the smaller the parallelization overhead and the better the performance in our new approach. According to our analyses, 2 KB and 4 KB blocks show some cache thrashing overheads due to the cache line mapping. Thus finding a good block size is important.

- Using 8-threads, it shows further improvements compared with the 4-threads run in many cases and does not show any major drawbacks. (Even the previous coarse-grain approach shows a small improvement using 8-threads compared with 4-threads as Table 1 shows.) Using 1-KB block, the performance improvement is furthered to 1.27~1.43-times speedup compared with the coarse-grain approach using 8-threads. This is because the cache misses appearing in block encryptions are masked off by the useful computations generated from 4 overloaded cores with 2 threads each.
- In the 8-threads run, the best throughput performance obtained for the previous coarse-grain approach and our new approach are 503.9 Mbps and 719.1 Mbps respectively (using 1 KB extended block size). The data size used is 16 MB. The new approach results in 1.43-times better throughput performance than the previous approach.

6.2 Results on GPU

We also conducted the same experiments on a GPU. We used Nvidia GeForce 8800 GT with 600 Mhz graphics clock, 1500 Mhz processor clock. It consists of 112 thread processors organized in 16 thread blocks. It has 16 KB shared memory per thread block. It doesn't have L1 data cache. The size of the device memory is 512 MB.

In the parallel implementation of our approach, we used CUDA. As explained earlier in Sect. 3.2, the CUDA model reflects the complicated memory hierarchy of the GPU. Depending on where the data is placed in the memory hierarchy, the resulting performance of the application varies significantly. The data placement is mostly under the programmer's control in CUDA. In the AES-CTR code, a significant portion of the run time is spent in the table lookup. Thus we store the 4 lookup tables in the shared memory in order to reduce the access overheads. Unlike the lookup tables, the plain text is stored in the global memory. Thus it is fetched from the global memory to the registers when a thread needs to encrypt the plain text into the cipher text. The pain text is read only once, thus we choose to store them in the global memory instead of the shared memory. Then we rely on the GPU's multithreading to hide the global memory access latency of threads with the useful computation cycles of other threads.

In order to compare our approach with the previous one, we first implemented the fine-grain and the coarse-grain parallelization approaches of the previous research [1]. Note that, on the GPU, the false-sharing effect of the cache block on a multi-core processor doesn't occur. Therefore, we implemented the fine-grain approach also. We use the shared-memory to store the lookup tables as in [1]. Figure 11 shows the performance results. In fact, the experiments closely reproduce the performance results in [1] when the shared memory is used to store the lookup tables. (In [1], they alternatively stored the lookup tables in the constant memory. The performance of the constant memory, however, is lower than the shared memory.) For small data sizes ($\leq$ 256 KB) the fine-grain approach performs better. As the data size increases, the coarse-grain approach outperforms the fine-grain approach. From this point on, we will use the coarse-grain results only to compare with our new approach.

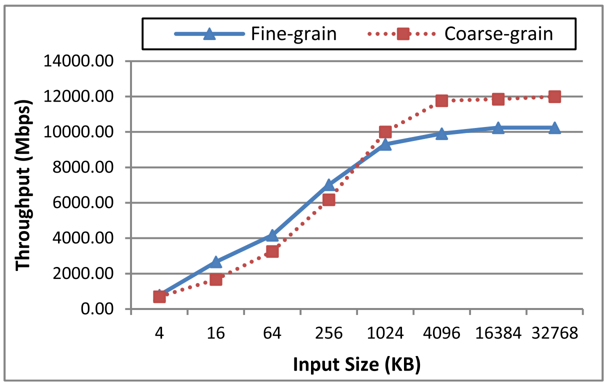


Fig. 11 Performance comparison of fine-grain approach and coarse-grain approach on Nvidia GeForce 8800 GT.

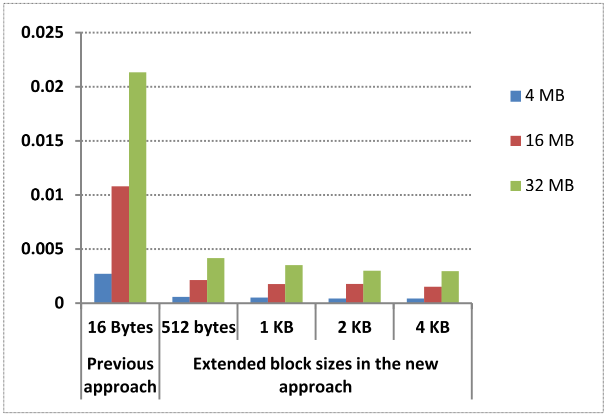


Fig. 12 Run time comparisons of new approach on GeForce 8800 GT with previous coarse-grain approach (16-byte).

Figure 12 shows the results of the new approach compared with the previous coarse-grain approach using 16-byte block size. The new approach shows significant performance improvements. Using the run times we compute the speedups of the new approach compared with the coarse-grain one. The speedups range 4.58~7.25 for different extended block sizes as Fig. 13 shows. In fact, on a GPU, the larger the block size, the larger the performance gain in general as expected in our new parallelization approach.

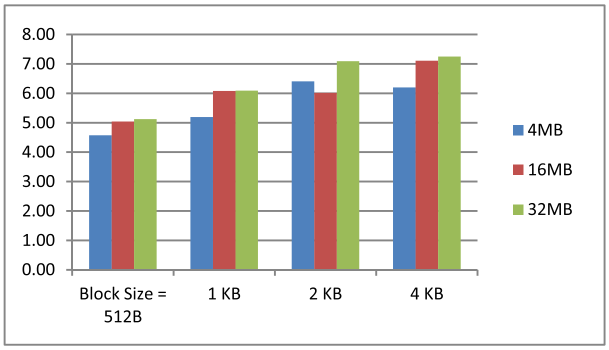


Fig. 13 Speedup of the new approach using various extended block sizes compared over the previous coarse-grain approach (16-bytes).

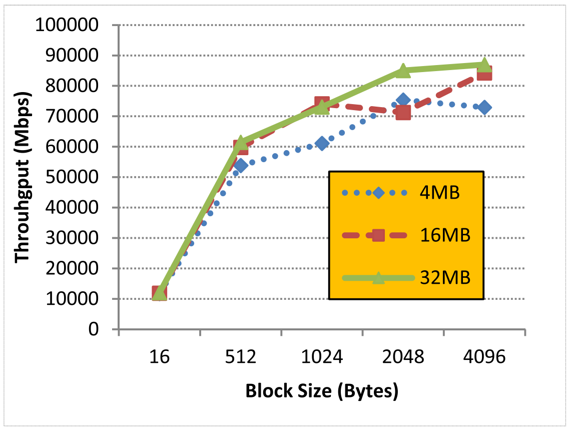


Fig. 14 Throughput of the new approach using various extended block sizes compared with the previous coarse-grain approach (16-bytes).

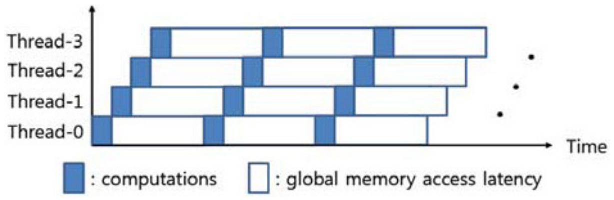
This is true for block sizes up to 4 KB. Considering that the speedup in Sect. 6.1 on a 4-core processor was in the range of 1.27~1.43 using up to 8-threads, the speedup on a GPU for the new approach is huge.

Figure 14 shows the throughput performance of the new approach compared with the previous approach. Using 16-bytes block of the previous approach, 12 Gbps throughput is achieved. Using the new approach, significantly higher throughput was achieved: 53 Gbps~87 Gbps. The highest throughput, 87 Gbps, is obtained when 32 MB data size was encrypted using 4 KB extended block. This is 7.25-times higher than the throughput of the previous approach (12 Gbps).

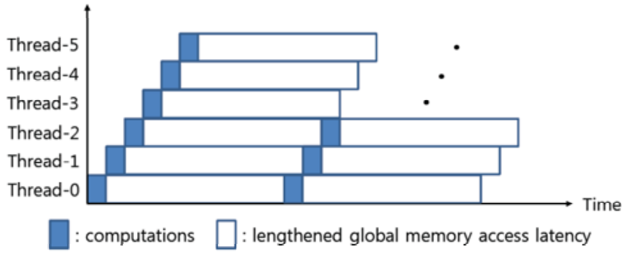
In presenting our performance results, we do not consider the data transfer time from the host (or CPU) memory to the device memory. In [1] with which we compared the performance of our approach, they do not include the data transfer time either. If we include the transfer time, the overall throughput performance will drop. The data transfer can be fully or partially overlapped with the computations (data encryptions). However, we couldn't implement the overlapping of the data transfer with the computations in our experiments, because the GPU we used (Nvidia GeForce 8800 GT) has a low CUDA Compute Capability (1.1) where

Table 2 Data transfer time for different block sizes.

Data Size (MB)	Transfer time (sec)
4	0.00180448
16	0.00703747
32	0.01371405



(a) Global memory access latencies effectively hidden with multithreading



(b) Lengthened global memory access latencies due to excessive degree of multithreading

Fig. 15 Performance effects of multithreading.

the overlapping is not available. Table 2 shows the data transfer time for different block sizes.

6.3 Effects of Extended Block Size on GPU Performance

The new approach using the extended block size significantly improves the performance as expected. This effect is more distinguished in the GPU results. Compared with the results on 4-core Intel processors, the observed speedup on the GPU is much larger (7.25-times vs. 1.43-times). The extended block size in our approach has the following positive performance effects on the GPU architecture and the memory system:

- As explained in Sect. 3.2, the GPU is executing in both the SIMD mode and the multithreaded mode. Having multiple threads available for execution can theoretically tolerate the long global memory access latencies which take a long time (≥ 200 cycles).
- The bandwidth to the global memory, however, has a limit. If there are too many threads accessing the global memory concurrently, it can lead to congestions in the global memory access paths and further lengthen the global memory access latencies [18].
- Figure 15 (a) depicts the case where an appropriate number of threads are used to effectively mask off the global memory access latencies by multi-threaded execution of the GPU. Figure 15 (b) depicts the case where

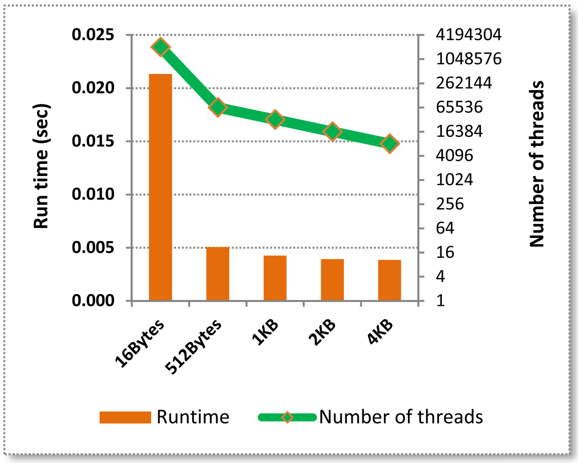


Fig. 16 Performance trends between the block size and the number of threads for 32 MB data size.

an excessive number of threads are generated which results in the lengthened global memory access latencies due to the conflicts on the global memory access paths by the generated threads.

- Finding an optimal number of threads to effectively hide the global memory access latency while efficiently utilizing the bandwidth is crucial for high performance. The number of threads is directly related to the block size for a given data size: the larger the number of threads, the smaller the block size as the total data size is fixed.
- Compared with the coarse-grain approach using 16-byte unit block, our approach results in a smaller number of threads by extending the block size for the given data. Figure 16 shows the performance trend between the run time and the number of threads for the 32 MB data using different block sizes 16-bytes, 512-bytes, 1 KB, 2 KB, 4 KB. As we extend the block size, the number of threads decreases and the performance improves for block sizes up to 4 KB. 8 KB block gives worse performance than 4 KB. (4 MB and 16 MB data sizes show similar trends, too, although they are not shown here.) Therefore, our extended block size approach efficiently utilizes the GPU's multithreading capability and leads to significant performance improvements.

7. Conclusion

In this paper, we proposed a new parallelization approach for a standard data encryption/decryption algorithm, AES-CTR. The proposed approach parallelizes the AES-CTR by extending the data block size encrypted at one time, thus significantly reducing the overheads incurred with the parallelization such as the number of procedure calls, and the parallel job scheduling and the synchronization overheads. Experimental results on a 4-core, 2.2 Ghz Intel processor with 2 GB DRAM, running Centos 5.5 OS shows that the new approach achieves up to 1.43-times speedup compared with the original coarse-grain approach where a sequence of 16-byte unit blocks are encrypted independently by multiple threads. The same experiments were also conducted on the Nvidia GeForce 8800 GT GPU with the code parallelized using CUDA. The new approach leads to 7.25-times speedup and the throughput performance improvement compared with the previous coarse-grain parallelization approach on the same GPU. The resulting throughput performance reaches up to 87 Gbps. Compared with the previous coarse-grain approach the new approach using the extended block leads to a more efficient use of the multithreading capability of the GPU and the global memory bandwidth. Thus it significantly improves the performance and the degree of the performance improvement on the GPU is much larger than the improvement on the multi-core processor.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the ministry of Education, Science, and Technology (Grant No: 2009-0089793 and 2012-042267).

References

[1] A.D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi, "Design of a parallel AES for graphics hardware using the CUDA framework," Proc. 2009 IEEE International Symposium on Parallel & Distributed Processing, May 2009.
[2] J.W. Bo, D.A. Osvik, and D. Stefan, "Fast implementation of AES on various platforms." Cryptology ePrint Archive, Report 2009/501, Nov. 2009, http://eprint.iacr.org/.
[3] J. Daemen and V. Rijmen, The Design of Rijndael: AES The Advanced Encryption Standard, Springer-Verlag, 2002.
[4] J. Daemen and V. Rijmen, "AES proposal rijndael [EB OL]," http://www.daimi.au.dk/˜ivan/rijndael.pdf, Oct. 2010.
[5] L. Deri, "nCap: Wire-speed packet capture and transmission," IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services, 2005.
[6] M. Dworkin, "Recommendation for block cipher modes of operation," NIST Special Publication 800-38A, 2001.
[7] F. Fusco and L. Deri, "High-speed network traffic analysis with commodity multi-core system," http://svn.ntop.org/imc2010.pdf
[8] K. Fatahalian and M. Houston, "A closer look at GPUs," Commun. ACM, Oct. 2008.
[9] Harrison and J. Waldron, "AES encryption implementation and analysis on commodity graphics processing units," CHES, ser. Lect. Nodes Comput. Sci., pp.209–226, 2007.
[10] O. Harrison and J. Waldron, "Practical symmetric key cryptographic on modern graphics hardware," 17th USENIX Security Symposium. San Jose, CA, Aug. 2008.
[11] B. He, N. Govindaraju, Q. Luo, and B. Smith, "Efficient Gather and Scatter Operations on Graphics Processors," Proc. SuperComputing 07, pp.175–186, Nov. 2007.
[12] S.A. Manacski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography," IEEE International Conference on Signal Processing and Communication, Nov. 2007.
[13] National Institute of Standards and Technology (NIST), "FIPS-197: Advanced Encryption Standard," http://www.itl.nist.gov/fipspubs/, Nov. 2001.

[14] "Nvidia gtx280", http://kr.nvidia.com/object/geforce_family_kr.html
[15] "Nvidia CUDA", http://developer.nvidia.com/object/cuda.html
[16] M. Pharr and R. Fernando, GPU Gems 2, Addison Wesley, 2004.
[17] M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw Hill, 2004.
[18] R.H. Saavedra-Barrera, D.E. Culler, and T. von Eicken, "Analysis of multithreaded architectures for parallel computing," ACM Symposium on Parallel Algorithms and Architectures - SPAA, pp.169–178, 1990
[19] L. Spracklen and S. Abraham, "Chip MultiThreading: Opportunities and challenges," 11th International Symposium on High-Performance Computer Architecture (HPCA-11), pp.248–252, 2005.
[20] V. Volkov and J.W. Demmel, "Benchmarking GPUs to tune dense linear algebra," Proc. SuperComputing 08, pp.Art. 31:1–11, Nov. 2008.

Nhat-Phuong Tran received the B.S. in Information Technology from Natural Science University, Vietnam in 2004, M.S. in Computer Science and Engineering, Myongji University, Republic of Korea, in 2012. He is now a Ph.D. student in the Dept of Computer Science and Engineering, Myongji University. His research interests are computer network and high performance computing.

Myungho Lee received his B.S. in Computer Science and Statistics from Seoul National University, Korea, M.S. in Computer Science, Ph.D. in Computer Engineering from University of Southern California, USA. He was a Staff Engineer in the Scalable Systems Group at Sun Microsystems, Inc, Sunnyvale, California, USA. He is currently an Associate Professor in the Dept of Computer Science and Engineering at Myongji University. His research interests are high performance computing architecture, compiler, and applications, with special interest in GPU computing.

Sugwon Hong earned BS in physics at Seoul National University, MS and Ph.D. in computer Science at North Carolina State University respectively. His professional experiences include Korea Institute of Science and Technology (KIST), Energy Economics Institute (KEEI), SK Energy Ltd. and Electronic and Telecommunication Research Institute (ETRI), all in Korea. Currently he is a professor at Dept. of Computer Science and Engineering, Myongji University since 1995. His major research fields are network protocol and architecture, network security.

Seung-Jae Lee received his B.S. and M.S. degrees in Electrical Engineering from Seoul National University, Korea and Ph.D. from the University of Washington, Seattle, USA. Currently, he is a professor at Myongji University and also a director of Next-generation Power Technology Center (NPTC). His primary research areas are protective relaying, distribution automation and substation automation.