The 14th IEEE International Conference on Computational Science and Engineering

Parallel Execution of AES-CTR Algorithm Using Extended Block Size

Nhat-Phuong Tran, Myungho Lee*, Sugwon Hong, Seung-Jae Lee†
Dept of Computer Science and Engineering, †Dept of Electrical Engineering
Myong Ji University, 38-2 San Namdong, Cheo-In Gu
Yong In, Kyung Ki Do, Korea 449-728
myunghol@mju.ac.kr

Abstract—Data encryption and decryption are common operations in a network based application programs with security. In order to keep pace with the input data rate in such applications, real-time processing of data encryption/decryption is essential. For example, in an environment where a multimedia data is streamed, high speed data encryption/decryption is crucial. In this paper, we propose a new approach to parallelize AES-CTR algorithm by extending the size of the block which is encrypted at one time across the unit block boundaries. The proposed approach leads to significant performance improvements using a general-purpose multi-core processor and a Graphic Processing Unit (GPU) which become popular these days. In particular, the performance improvement on GPU is dramatic; close to 9-times faster compared with the original coarse-grain parallelization approach, mainly thanks to the "multi-core" nature of the GPU architecture.

Keywords-AES;multi-core;GPU; parallelization

I. Introduction

are becoming increasingly popular. Applications based on multimedia data streaming are good examples of such applications. 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 [9]. It is widely used recently because of its high security and low cost. For encryption and decryption of multiple blocks, it has several modes of operations such as CTR mode [2, 3, 5] which is used in this paper for parallelization. In an application program based on multimedia data streaming, the data is received continuously with high input rate. In order to keep pace with the high data input rate, real-time processing of data encryption and decryption are crucial.

Recently, incorporating multiple processor cores on a single chip (or multi-core processor) has become a main stream microprocessor design trend since mid-2000. 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) [15]. All the CPU vendors including Intel, AMD, IBM, Oracle/Sun, among others have introduced their multi-core processors in the market. The multi-core design is also adopted in embedded systems such as ARM11 MPCore (quad-core) processor.

* Corresponding author

The Graphic Processing Unit (GPU) introduced in the late 1990s as a co-processor for accelerating the simulation and visualization of 3D images is also commonly used these days. In the earlier GPU architectures, there were separate processing units for Shader, Vertex, Pixel. In the latest GPU's, those units are, however, incorporated into multiple uniform programmable processing units in recent GPU's which have made huge floating-point performance improvements possible [10, 11, 12]. Furthermore, in order to utilize the flexible hardware design, user friendly programming environments are recently developed by the GPU vendors such as CUDA from NVidia, OpenCL from AMD/ATI. Using those environments along with the flexible GPU architecture has led to innovative performance improvements in many application areas, and many more to come [13, 16].

In this paper, we develop a new parallelization technique to speedup the AES-CTR algorithm 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 applied. This approach, thus, reduces the overheads associated with procedure calls and parallel job scheduling incurred with the execution of AES-CTR. By implementing the proposed parallelization technique on a general-purpose multi-core processor (2.2Ghz 4-core Intel processor) and a GPU (NVidia GT9500), we've observed a significant performance improvements compared with a previous coarse-grain parallel approach. In fact, the GPU shows a dramatic performance improvement, close to 9-times faster perforrmance, than the coarse-grain approach. This huge gain is mainly thanks to the "multi-core" nature of the GPU architecture.

The rest of the paper is organized as follows: Sections II gives an overview of AES algorithm and its modes of operations. Section III shows the architecture of the latest general-purpose multi-core processor and multi-core based GPU architectures, and their programming models. Section IV explains our parallelization technique compared with the previous approaches employing fine-grain and coarse-grain approaches. Section V shows the experimental results of the new approach compared with the previous one on a 4-core Intel processor and NVidia GT9500 GPU. Section VI wraps up the paper with conclusion.

II. Overview of AES algorithm

The Advanced Encryption Standard (AES) is a symmetric cryptographic algorithm published by NIST [9] which had replaced the previous Data Encryption (DES) standard. AES is the most widely used block cipher in recent years because of its high security and low cost. In order to encrypt and decrypt more than one block, modes of operation have been developed. In this section, we describe the main computation steps for Advanced Encryption Standard (AES) block cipher algorithm first and then describe its modes of operation.

A. Computation Step

AES algorithm is carried out by applying a number of repetitions of transformation rounds that converts the input plain text into the final cipher text. AES has a fixed block size of 128 bits with three key lengths 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 various: 10 for 128 bits, 12 for 192 bits and 14 for 256 bits. Each round includes several steps. The output of each round is 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 added and for the last round where the last step is skipped [2, 3, 5].

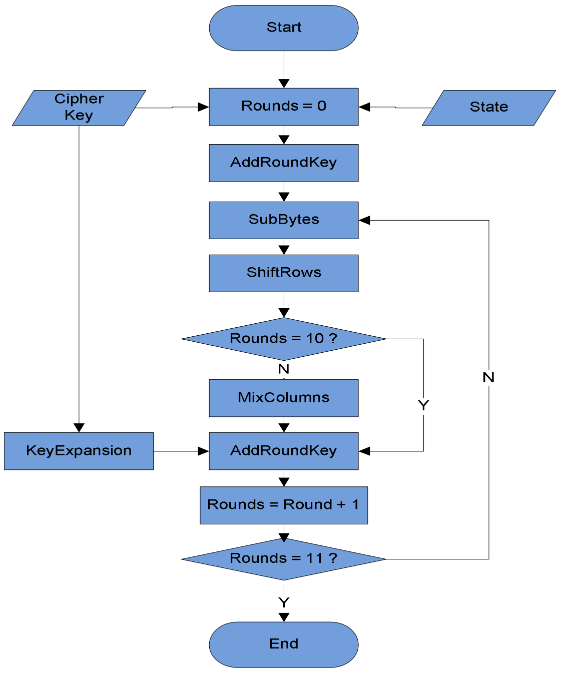

Figure 1. AES-128 algorithm

Figure 1 above shows the step of AES algorithm; we can see that the AES-128 algorithm is iterative with 10 rounds. The input to the algorithm is a block of 128 bits plain text which is represented by a 4x4 byte matrix called "State".

- KeyExpansion is used to generate the RoundKeys from the original key for rounds.
- The four round stages are AddRoundKey (XOR each column of the State with a word from 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)
- For the initial round, perform only the operation AddRoundKey.
- For the next N–1 rounds, perform four operations SubBytes, ShiftRows, MisColumn and AddRoundKey.
- For the last round, perform the same operation the previous N – 1 rounds except without the MixColumns operation.

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

There are two general computations in AES algorithm [2]. One is the matrix computation with operations AddRoundKey, SubBytes, ShiftRows, and MixColumns applied in each round. The other is the table lookup. With table lookup method, the different steps of the round can be combined in a single set of table lookups.

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

Where, a refers to the input matrix variable, e 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 for 4 KB of storage space and obtained through combination as follows:

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

$$T_2[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \end{bmatrix} \quad T_3[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \end{bmatrix}$$

, where $\bullet$ is a GF(2^8) finite field multiplication [3]. In this paper, we use the second method to implement.

B. Modes of Operations

There are five modes of operation for AES [5]:

1) Electronic codebook mode (ECB mode)
2) Cipher block chaining mode (CBC mode)
3) Output feedback mode (OFB mode)
4) Counter mode (CTR mode)
5) Cipher feedback mode (CFB mode)

In this section we describe CTR mode only because CTR mode are parallel in nature and secure by using different keys in blocks. Thus we chose CTR mode in the paper. Details of the other modes can be found in [5].

In the CTR 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 $T_i = (\text{cter} + i - 1) \bmod 2^m$. Then encrypt the plain text blocks by $c_i = p_i \oplus E_k(T_i)$. Figure 2 below shows an example application of AES algorithm.

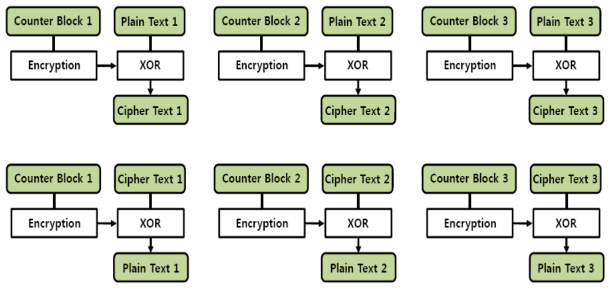

Figure 2. Example application of AES-CTR mode

III. OVERVIEW OF ARCHITECTURE

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

A. 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 density. Instead of employing a complicated processor pipeline on a chip with an emphasis on improving single 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) [15]. All the CPU vendors including Intel, AMD, IBM, Oracle/Sun, among others have introduced multi-core processors in the market. The multi-core design is also adopted in embedded systems such as ARM11 MPcore (quad-core) processor based systems introduced lately.

In addition to the CMP based multi-core design, some designs go one step further to incorporate Simultaneous MultiThreading (SMT) or similar technologies on a processor core. Examples are Intel Nehalem and UltraSPARC T2/T3 microprocessor from Oracle/Sun. 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 or off the processor chip. Each core also has *M* hardware threads performing SMT or similar features. 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 x 8) threads can execute on a chip at the same time. Each core has 8KB private data cache. The level-2 unified cache is 4MB in size.

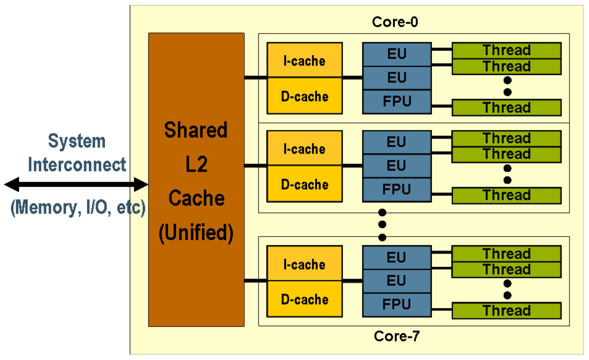

Figure 3. Architecture of an advanced multi-core processor

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 on the same chip. Software processes or threads running on the cores of the same processor chip compete for the shared resources, which can cause conflicts and hurt performance. Thus exploiting the full performance potential of multi-core processors is a challenging task [15].

B. 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. The first GPU was introduced in the market in 1999: NVidia GeForce 256. Since then GPU has become widespread and these days it is incorporated in almost all Desktop Computers. The clock rate of the latest GPU has ramped up to 675Mhz compared with 120Mhz in the earlier models. Furthermore it has shown impressive improvement in the floating-point performance, far exceeding that of the latest CPU's and the performance gap is widening.

In the architecture of earlier GPU's, there were separate processing units for Shader, Vertex, Pixel. In the latest GPU's, those units are incorporated into multiple programmable processing units or thread processors (see Figure 4) [10]. This design trend reflects the fact that application programs dealing with 3D graphic images are suitable for SIMD (Single Instruction Multiple Data) processing. In order to utilize the flexible hardware design, more user friendly programming environments are recently developed by the GPU vendors. CUDA from NVidia, OpenCL from AMD/ATI are good examples of such software environments [11, 12]. Using those environments, programmers can have more direct control over the GPU pipeline and memory hierarchy, whereas in the old GPU's they relied on specific graphics API's. The flexible GPU hardware and user friendly software have recently led to a number of innovative performance improvements in many application areas and more improvements are still to come [13, 16].

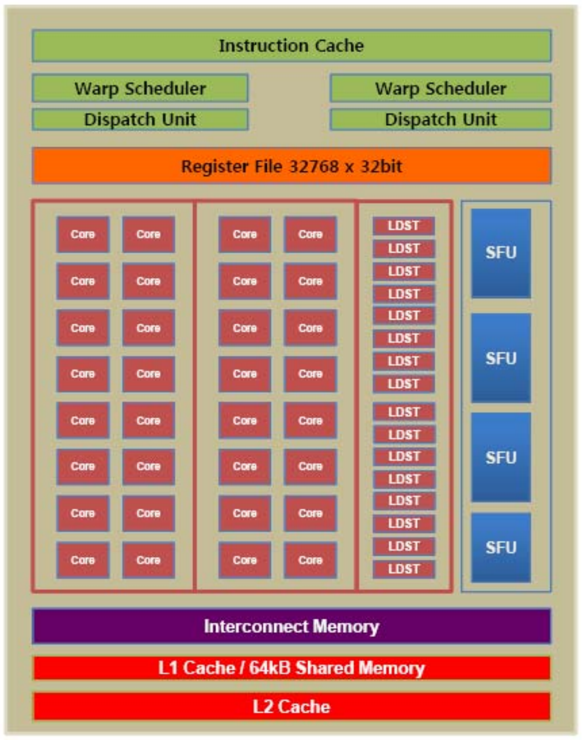

Figure 4. Architecture of NVidia GTX480 GPU

In the experiments conducted in the paper, we use 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 Global memory accessed from the thread block arrays [11, 12]:

- Global memory is an off-chip GDDR of which the size ranges from 256MB to 4GB. Through the Global memory GPU can communicate with the host CPU.
- Shared memory sits within each thread block. Its typical size is 16KB. The access time closely matches with the register access time, thus it is a very fast memory.
- Registers are used for temporarily storing data used for GPU computations, similar to CPU registers.

In CUDA programs, data needed for computations on GPU is transferred from the host memory to the Global memory on the GPU, distributed to the shared memories by the programmer, then used by thread blocks and threads. Therefore, GPU computations are suitable for parallel executions of the same instruction on a large block of data rather than performing various computations on a small amount of data.

IV. PARALLELIZATION OF AES ALGORITHM

In a typical network-based application with security, data encryption and decryption are intensively performed with respect to large amounts of data continuously received from and forwarded to other senders/receivers. In such an environment, the demand for parallel execution of AES is high. In this Section, we first describe two previous parallelization methodologies for AES-CTR. Then we introduce our new parallelization approach using an extended block size.

```
aes_ctr (iblock, oblock, iv, key, r, tb0, tb1, tb2,tb3)
begin
            // Parallelize the loop below (coarse-grain)
     for(i = 0, i < TOTAL_SIZE , i+=16)
          encrypt_block(iblock+i, oblock+i, iv+i, key, r, tb0,tb1, tb2, tb3)
     end for
end

encrypt_block(iblock, oblock, iv, key, r, tb0, tb1, tb2, tb3)
begin
            byte i, j , rounds;
            byte temp[16],c[16];

            // Parallelize the loop below (fine-grain)
            for( i=0 ; i<16 ; i++ )
                        c[i] = iv[i] ^ key[i];

            for(rounds=1, rounds<=r, rounds++)
            // Parallelize the loop below (fine-grain)
                        for( j=0, j<4 , j++ )
                           for( i=0 , i<4 , i++ )
                                   temp[(i<<2)+j] = \
                                       (tb0[(i<<8)+c[0+j]]) \
                                   ^ (tb1[(i<<8)+c[4+((j+1)%4)]]) \
                                   ^ (tb2[(i<<8)+c[8+((j+2)%4)]]) \
                                   ^ (tb3[(i<<8)+c[12+((j+3)%4)]]) \
                                   ^ (key[(rounds<<4)+(i<<2)+j])
                           end for
                        end for
                        // Parallelize the loop below (fine-grain)
                        for( j=0 ; j<16 ; j++ )
                                   c[j] = temp[j]
                        end for
            }
            // Parallelize the loop below (fine-grain)
            for( i=0, i<16, i++ )
                        oblock[i] = c[i] ^ iblock[i]
            end for
end
```

Figure 5. Fine/coarse-grain parallelization of major routines in AES-CTR using 16-bytes block size

A. *Previous Approaches*

As described in the previous Section, data encryption in AES-CTR goes through a number of rounds with respect to a unit sized block. Within each round, four computation steps involving XOR, byte substitution, shift, etc., are performed with respect to each block. A straightforward approach parallelizes the execution of four computation steps. In Figure 5 above, fine-grain approach parallelizes the for-loops in procedure encrypt_block (also see Figure 9 on page 8). This approach, however, incurs large parallelization overheads such as synchronization and multiple concurrent accesses to the same cache block which can lead to false-sharing. Thus this approach is not a desirable one.

Contrast to the fine-grain approach, a coarse-grain approach attempts to parallelize the AES-CTR algorithm at the level of unit-sized block. Since a large amount of data is typically received at a computing node in a network-based application, there are multiple blocks in the received data. The coarse-grain approach executes the AES encryptions for multiple blocks using different threads. In Figure 5 above, coarse-grain parallelizes the for-loop in procedure aes_ctr, but does not parallelize the for-loops in encrypt_block. For example, assume that N-blocks are received in total and 4-threads are used for the encryption. 4-blocks are encrypted at the same time, and the execution is repeated N/4-times (see Figure 10 on page 8). The coarse-grain approach, thus, does not attempt to speed-up a single block encryption, but attempt to divide the total number of blocks amongst a number of threads. Thus attempts to exploit task-level parallelism. This approach may lead to a longer run time to encrypt a single block, but incurs significantly low parallelization overheads such as synchronization compared with the fine-grain approach. Furthermore, it significantly reduces the risk of false-sharing. Therefore this approach gives better performance than fine-grain approach in encrypting N-blocks [1].

B. *Our New Approach*

Given N-blocks, the coarse-grain approach distributes N/P-blocks to each core where P is the number of cores available for parallel execution of AES. Each processor core, thus, encrypt the assigned blocks sequentially N/P-times. The N/P-times repetition usually involves procedure calls and parallel job scheduling overheads. In our approach, we significantly reduce these overheads by extending the block size across the unit block boundaries. For instance, creating a bigger block (e.g., 1024Bytes) by coalescing 64 16-Bytes unit blocks. With extended block size, evidently, we reduce the cost of separating blocks to each P. In addition, each P can be used more effectively because the sum of the work which it processes increases. This approach, however, needs to extend the key size to the length of the extended block size so that the encryption of 1024-Bytes can be performed at one time. In order to implement this, we need to create 1024-Bytes array and replicate the 16-Bytes key 64-times, for example. (See Figure 6, 11 on page 8 for more details of the approach.)

```
aes_ctr (iblock, oblock, iv, key, r, tb0, tb1, tb2,tb3)
begin
            // Parallelize the loop below, with enlarged block size
     for(i = 0, i < TOTAL_SIZE , i+= NEW_BLOCK_SIZE)
        encrypt_block(iblock+i, oblock+i, iv+ i, key, r, tb0,tb1, tb2, tb3)
     end for
end

encrypt_block(iblock, oblock, iv, key, r, tb0, tb1, tb2, tb3)
begin
            byte i, j , rounds;
            byte temp[NEW_BLOCK_SIZE], c[NEW_BLOCK_SIZE];

            for( i=0 ; i< NEW_BLOCK_SIZE; i++ )
                        c[i] = iv[i] ^ key[i];
            end for

            for(rounds=1, rounds<=r, rounds++)
                        for (k=0 ; k < NEW_BLOCK_SIZE ; K += 16)
                        for( j=0, j<4 , j++ )
                            for( i=0 , i<4 , i++ )
                              temp[k + (i<<2)+j] = \
                                          (tb0[(i<<8)+c[k+ 0+j]])  \
                                      ^ (tb1[(i<<8)+c[k+ 4+((j+1)%4)]]) \
                                      ^ (tb2[(i<<8)+c[k+ 8+((j+2)%4)]]) \
                                      ^ (tb3[(i<<8)+c[k+ 12+((j+3)%4)]]) \
                                      ^ (key[(rounds<<4)+(i<<2)+j])
                                                end for
                                      end for
                        end for
                        for( j=0,  j<NEW_BLOCK_SIZE , j++ )
                                      c[j] = temp[j]
                        end for
            }
            for( i=0, i< NEW_BLOCK_SIZE, i++ )
                        oblock[i] = c[i] ^ iblock[i]
            end for
end
```

Figure 6. Parallelization of AES-CTR using extended block size

V. Experimental Results

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

TABLE I. Performance results on 4-core Intel processor

Data Size (MB)	Coarse-Grain Parallelization Run Time (Sec)			New Approach (Sec) Block Size = 1KB		
	1 threads	4 threads	8 threads	1	4	8
4	0.248	0.063	0.0645	0.2	0.051	0.049
16	1.04	0.263	0.254	0.801	0.212	0.178
32	1.99	0.505	0.508	1.56	0.395	0.4

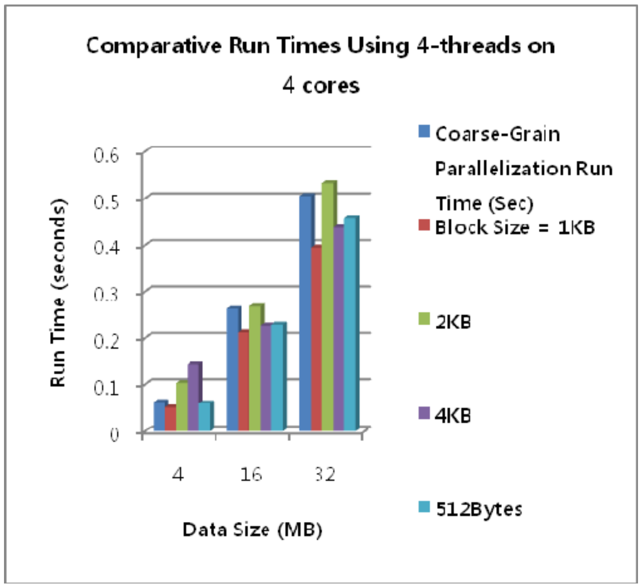

Figure 7. Performance comparisons of the coarse-grain approach and the new approach on a 4-core Intel

A. *Results on General-Purpose Multi-Core Processor*

We've parallelized the AES-CTR code for both the coarse-grain approach and the new approach using OpenMP [14]. Experiments were conducted on 2.2Ghz, 4-core Intel Core 2 Duo processor with 2GB DRAM, running Centos 5.5 OS. The run times of the two approaches were measured. TABLE I above compares the run times of the coarse-grain approach and the new approach, using 1-,4-,8-threads on 4 CPU cores.

- Both the coarse-grain approach and the new approach show good scalability when comparing the run time using 1-thread and 4-threads. Using 8-threads might be an over kill for a 4-core processor. However, in many cases, it shows small improvements compared with the 4-threads run and does not show major drawbacks.
- Using 1KB block size, extended from 16Bytes by 64-times, the new approach shows 1.25~1.28x speedup compared with the coarse-grain approach, as the following Figure 7 above shows. The performance improvements are almost uniform for different data sizes (4MB, 16MB, 32MB).
- The best block size turns out to be 1KB. 2KB block size, however, shows worse performance than the original coarse-grain approach. 512Bytes and 4KB block sizes are also showing some improvements for a larger data sizes such as 16MB and 32MB. But no improvements for a small data size such as 4MB. This is somewhat out of our expectation, because it is speculated that the larger the block the smaller the parallelization overhead in our new approach. Thus finding a good block size is important.

TABLE II. Performance results on NVidia GT9500

Size (MB)	16-Bytes Block	Block Size = 1KB	2KB	4KB	512Bytes
4	0.04688	0.0074	0.006	0.0062	0.0084
16	0.1877	0.025	0.026	0.022	0.031
32	0.3754	0.0524	0.045	0.044	0.0623

B. *Results on GPU*

We also conducted the same experiments on a GPU. We used CUDA to parallelize the AES-CTR code for both coarse-grain approach and the new approach. Above TABLE II shows the experimental results of both approaches. Also, Figure 8 on page 8 shows the speedup's of the new approach using extended block sizes compared with the original approach using 16Bytes block size.

- The new approach shows dramatic performance improvements on a GPU. TABLE II above shows the speedup of the new approach, for different block sizes. The speedup ranges from 6.03~8.53. This leads to an impressive throughput of 5.82Gbps!
- Considering that the 4-core speedup was in the range of 1.25~1.28x, GPU's speedup for the new approach is huge. Our initial analysis shows that the "multi-core" nature of the GPU architecture is the main contributor for the dramatic speedup. More detailed analyses need to be done.
- Unlike on the general-purpose multi-core processor, larger block sizes give more speedup than small blocks. This was quite expected when we designed the new parallelization approach using the extended block size, as the bigger blocks reduce the number of iterations in the main routine "aes_ctr" in Figure 6 which reduces the procedure call overheads and job scheduling overheads incurred with parallelization.

TABLE III. Speedup of the new approach compared with the coarse-grain approach

Size (MB)	Block Size = 1KB	2KB	4KB	512Bytes
4	6.34	7.81	7.56	5.58
16	7.51	7.22	8.53	6.05
32	7.16	8.34	8.53	6.03

VI. CONCLUSION

In this paper, we proposed a new parallelization approach for data encryption/decryption application, AES-CTR. The proposed approach parallelize the AES-CTR by extending the data block size encrypted at one time, thus reducing the overheads incurred with the procedure calls and the parallel job scheduling. Experimental results on a 4-

core, 2.2Ghz Intel processor with 2GB DRAM, running Centos5.5 OS shows that the new approach achieves 1.25~1.28x speedup compared with the original coarse-grain approach where a sequence of 16-Bytes blocks are encrypted independently by multiple threads. Same experiments were conducted using NVidia GT9500 GPU using CUDA. The speedup on a GPU is dramatic; it showed 6.03~8.53x speedup compared with the original approach run on the same GPU. The large speedup attributes to the "multi-core" nature of the GPU architecture. Furthermore, the speedup is larger on the GPU with a larger block size as the number of procedure calls and the parallel job scheduling overheads decrease. More analyses and experiments are planned for furthering the improvement using an extended block size both on a GPU and a general-purpose multi-core processor.

ACKNOWLEDGMENT

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).

REFERENCES

[1] A.D. Biagio, A. Barenghi, G. Agosta, G. Pelosi, "Design of a Parallel AES for Graphics Hardware using the CUDA framework", Proceedins of the 2009 IEEE International Symposium on Parallel & Distributed Processing, May 2009.

[2] J. Daemen, V. Rijmen, "The Design of Rijndael: AES The Advanced Encryption Standard", Springer-Verlag, 2002.

[3] J. Daemen, V. Rijmen, "AES Proposal Rijndael [EB OL]", http://www.daimi.au.dk/~ivan/rijndael.pdf, Oct 2010.

[4] L. Deri, "nCap: Wire-speed packet capture and transmission," the IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services 2005.

[5] Morris Dworkin, "Recommendation for Block Cipher Modes of Operation",

[6] F. Fusco and L. Deri, "High-Speed Network Traffic Analysis with Commodity Multi-Core System," http://svn.ntop.org/imc2010.pdf.

[7] K. Fatahalian, M. Houston, "A Closer Look at GPUs", Comm. of ACM, Oct 2008.

[8] B. He, N. Govindaraju, Q. Luo, B. Smith, "Efficient Gather and Scatter Operations on Graphics Processors", Proceedings of the SuperComputing 07, pp. 175-186, Nov 2007.

[9] National Institute of Standards and Technology (NIST), "FIPS-197: Advanced Encryption Standard." http://www.itl.nist.gov/fipspubs/ , Nov. 2001.

[10] "NVidia gtx280", http://kr.nvidia.com/object/geforce_family_kr.html

[11] "NVidia CUDA", http://developer.nvidia.com/object/cuda.html

[12] Matt Pharr et. al., "GPU Gems 2", Addison Wesley, 2004.

[13] NVidia CUDA Programming Guide", http://kr.nvidia.com/object/cuda_develop_kr.html

[14] M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw Hill, 2004.

[15] L. Spracklen and S. Abraham, Chip MultiThreading: Opportunities and Challenges, 11th International Symposium on High-Performance Computer Architecture (HPCA-11), pp 248-252, 2005.

[16] V. Volkov and J.W. Demmel,"Benchmarking GPUs to Tune Dense Linear Algebra", Proceedings of the SuperComputing 08, pp. Art. 31:1-11, Nov 2008.

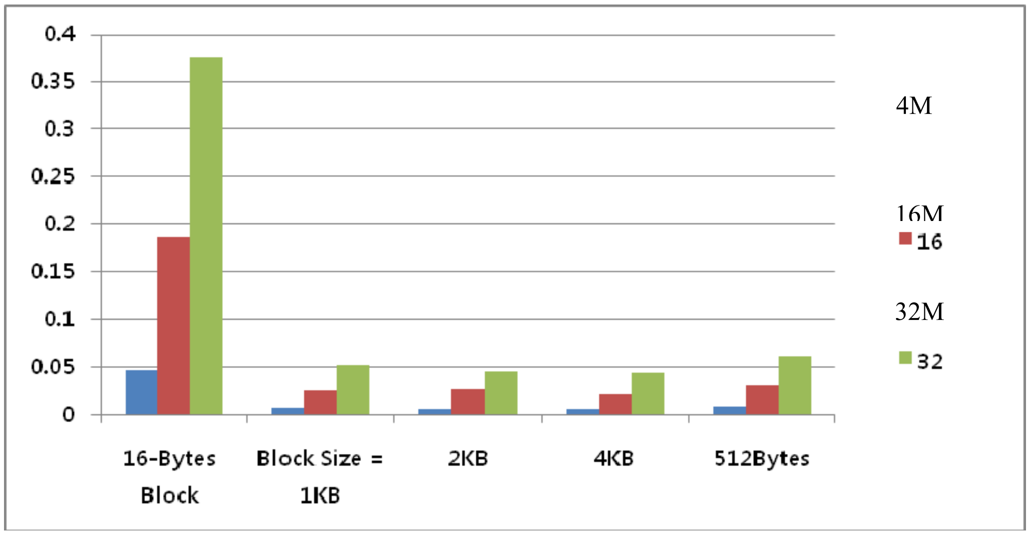

Figure 8. Performance comparisons of the coarse-grain approach and the new approach on NVidia GT9500

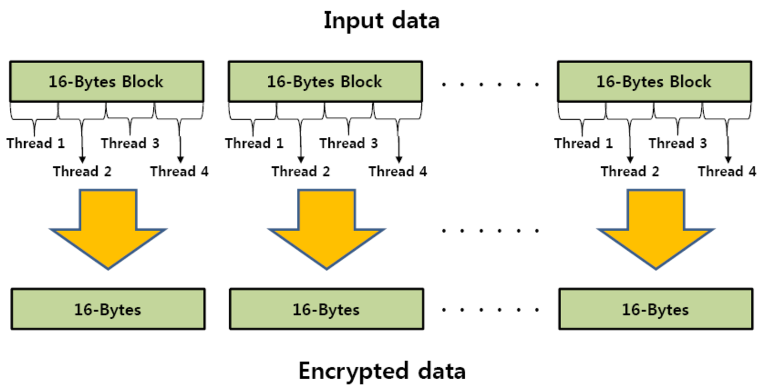

Figure 9. Fine-grain parallel encryption to a 16-Byte block, using 4-threads

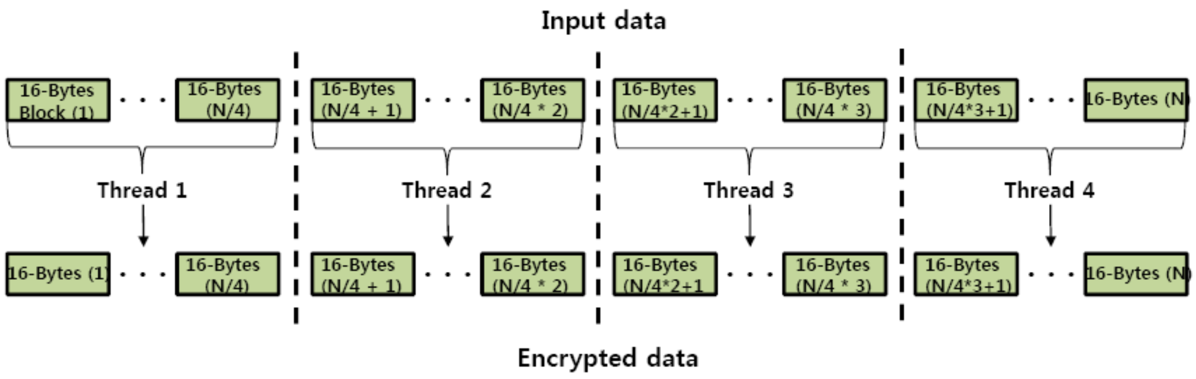

Figure 10. Coarse-grain parallel encryption using 16-Byte block size, 4-threads

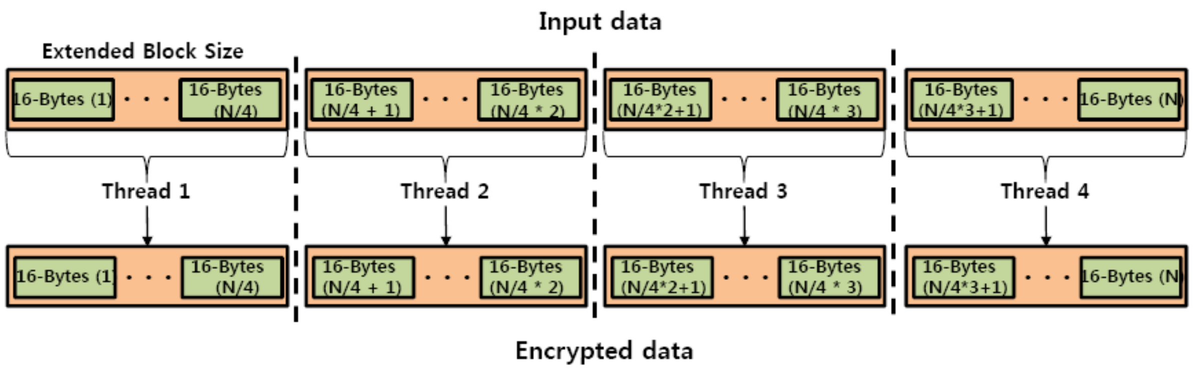

Figure 11. Encryption using extended block size