Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.25, NO.4, Aug. 2015 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2015.25.4.719

GPU의 스레드와 공유메모리를 이용한 LEA 최적화 방안*

박 무 규,† 윤 지 원‡
고려대학교 정보보호대학원

Optimization of Lightweight Encryption Algorithm (LEA) using Threads and Shared Memory of GPU*

Moo Kyu Park,† Ji Won Yoon‡
Center for Information Security Technologies, Korea University

요 약

최근 빅데이터와 클라우드 보안에 대한 관심이 증가함에 따라 이를 효율적으로 암호화하기 위해 경량화된 고속 암호에 대한 연구가 진행되어 왔다. 그 결과, 국가보안기술연구소에서는 경량·고속 블록 암호인 LEA를 개발하였다. 경량화 암호인 LEA를 효율적으로 암·복호화를 위해 CPU가 아닌 GPU를 이용한 고속화 연구들이 최근에 소개되었다. 그럼에도 불구하고, GPU사용에 있어서의 가이드라인에 대한 연구는 여전히 미흡하다. 본 논문에서는 LEA알고리즘이 대용량 처리를 위해 사용될 때, 효율적인 GPU를 활용한 LEA 최적화방안에 대해 제안한다.

ABSTRACT

As big-data and cloud security technologies become popular, many researchers have recently been conducted on faster and lighter encryption. As a result, National Security Research Institute developed LEA which is lightweight and fast block cipher. To date, there have been various studies on lightweight encryption algorithm (LEA) for speeding up using GPU rather than conventional CPU. However, it is rather difficult to explore any guideline how to manipulate the GPU for the efficient usage of the LEA. Therefore, we introduce a guideline which explains how to implement and design the optimal LEA using GPU.
Keywords: GPU, CUDA, Parallel Programming, LEA

I. 서 론

2000년대에 들어오면서 영화 및 게임 산업에서는 화려한 영상처리를 위해 뛰어난 성능을 가진 그래픽 카드를 요구하기 시작했다. 이는 자연스럽게 GPU (그래픽처리장치:Graphic processing unit)의 발전으로 이어졌다. 그러나 그래픽 전용 API함수로 인해 GPU의 성능을 일반적인 데이터 연산에 적용하는데 어려움이 많았다. 이러한 문제점을 해결하기 위해, 영상처리뿐만 아니라 일반적인 데이터 연산에서도 GPU의 성능을 활용할 수 있는 GPGPU (General Purpose Graphic Processing Unit: 범용그래픽처리장치)가 개발되었다. 현재 GPGPU는 신호처리, 물리 시뮬레이션, 재무예측 등 대용량 데이터 처리를 목적으로 한 분야에서 활용되고 있다. GPGPU의 활용을 극대화하기 위해 NVIDIA사에

접수일(2015년 4월 1일), 수정일(2015년 6월 16일), 게재확정일(2015년 6월 22일)
* 본 연구는 미래창조과학부의 지원으로 한국 연구 재단의 기초 과학 연구 프로그램의 일환으로 시행되었음(NRF-2013R1A1A1012797).
† 주저자, ctupmk@korea.ac.kr
‡ 교신저자, jiwon_yoon@korea.ac.kr(Corresponding author)

서는 그래픽카드에서의 병렬 프로그램 개발 환경을 제공하기 위해 CUDA (Compute Unified Device Architecture)를 개발하였다.

특히, 암호분야에서는 GPU를 활용하여 암호 알고리즘의 최적화 연구가 활발히 진행 되어왔다 [1-5]. 본 논문에서는 NVIDIA사의 GPGPU인 Tesla K20 과 CUDA를 이용하여 LEA (Lighted Encryption Algorithm)를 최적화하는 방법에 대해 설명하고자 한다.

II. 관련 연구

일반적으로 GPU는 많은 수의 코어와 그래픽API를 이용하여 영상처리에 중점적으로 이용되어 왔다. 하지만, NVIDIA사의 CUDA개발로 GPU에 최적화된 일반적인 연산이 가능하게 되었다. 본 문단에서는 CUDA와 이를 이용하여 최적화한 암호기술들에 대해 알아보고, 최신 블록암호 LEA 암호화 알고리즘에 대해 설명하고자 한다.

2.1 CUDA

CUDA는 Compute Unified Device Architecture의 약자로서, NVIDIA사에서 제작한 GPU를 병렬 연산이 가능하게 하도록 하는 병렬 프로그램 통합 개발 환경이다. CUDA는 GPU를 이용하여 범용적인 프로그램을 개발할 수 있도록, 프로그램 모델, 프로그램 언어, 컴파일러, 라이브러리, 디버거, 프로파일러를 제공한다. CUDA의 기본 프로그램언어는 C/C++기반이며, 최근에는 Java, Python을 통하여도 구현이 가능하다. 본 단계에서는 CUDA의 프로세서 아키텍쳐 및 메모리 구조에 대하여 설명하고자 한다[1,2].

2.1.1 CUDA 프로세서 구조

CUDA의 프로세서는 스트리밍 프로세서(SP: Streaming processor), 스트리밍 멀티 프로세서(SM: Streaming multi processor) 그리고 텍스처/프로세서 클러스터로 구성되어 있다.

2.1.1.1 스트리밍 프로세서(SP)

스트리밍 프로세서는 GPU 내부에서 실질적인 연산을 담당하는 코어 유닛이며, GPU의 종류마다 장착된 코어의 개수는 다양하다. Tesla K20의 경우 2496개의 스트리밍 프로세서가 장착되어 있다[3]. 스트리밍 프로세서는 데이터 연산을 위한 멀티 뱅크 레지스터와 실수 계산을 위한 FPU, 정수 계산을 위한 ALU, 그리고 데이터를 가져오고, 저장할수 있는 LSU로 구성된다. CUDA에서 스트리밍 프로세서는 4개의 스레드를 구동 시킬 수 있으며, 그 구조는 Fig.1 과 같다.

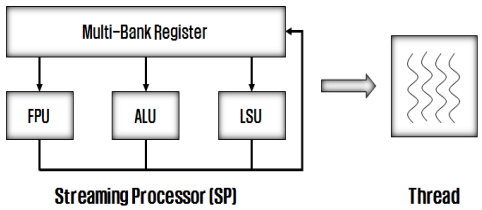

Fig. 1. Structure of SP

2.1.1.2 스트리밍 멀티 프로세서(SM)

스트리밍 프로세서 8개가 모여서 구성되는 구조로, 스트리밍 프로세서 포함하여, 명령어 캐시와 데이터 캐시를 장착하고 있다. 또한 스트리밍 멀티 프로세서는 데이터를 공유하는 공유 메모리와 I 캐시, C 캐시로 구성되어 있으며, 특수함수를 구현할 수 있는 SFU (Special Function Unit)을 가지고 있다. CUDA에서 스트리밍 멀티 프르세서는 블록을 의미하며, Fig. 2와 같다.

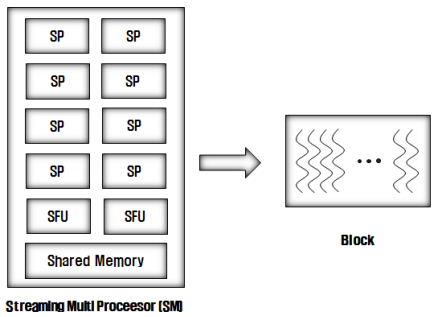

Fig.2. Structure of SM

2.1.1.3 텍스처/프로세서 클러스터(TPC)

텍스처/프로세서 클러스터는 스트리밍 멀티 프로세

서를 제어하고 그래픽 카드에서의 주요 기능인 텍스처와 지오메트리셰이더 처리기능을 가지고 있으며,그 구조는 Fig. 3과 같다.

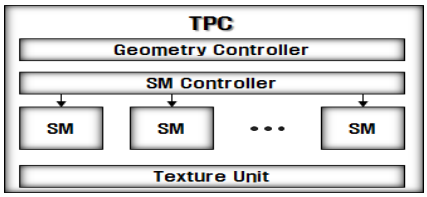

Fig. 3. Structure of TPC

2.1.2 CUDA 메모리 구조

CUDA의 메모리 구조는 오프칩 메모리와 온칩 메모리로 구성되어 있다 [8].

오프칩 메모리는 GPU 프로세서 밖에 있는 메모리를 의미하며, 글로벌 메모리 (Global memory), 상수 메모리 (Constant Memory), 텍스쳐 메모리 (Texture Memory)로 구성되어 있다. 오프 칩 메모리의 특징은 메모리 접근 속도는 느리지만, 접근에 제약사항은 없다.

온칩 메모리는 GPU 프로세서 내에 있는 메모리로, 레지스터(Register), 공유메모리(Shared memory), 스레드(Thread)로 구성되어 있다. 온칩 메모리의 특징은 오프칩 메모리의 특징과는 반대로 메모리 접근속도는 빠르지만, 접근에 제약사항이 많다.

온칩과 오프칩 메모리의 구조는 Fig. 4와 같다.

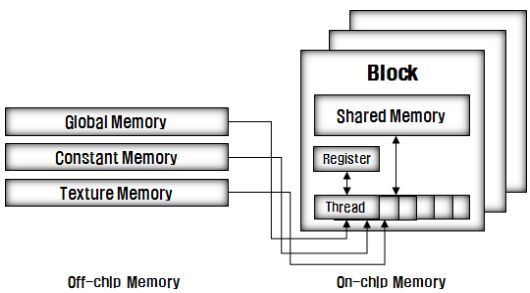

Fig. 4. Structure of CUDA Memory

2.2 CUDA를 이용한 암호 알고리즘 최적화

CUDA를 활용하여 암호 알고리즘을 최적화하는 방법에 대해서는 많은 연구가 진행되어 왔다. 그 대표적인 연구 사례가 2007년에 진행된 AES 알고리즘 최적화이다[4]. 이 연구에서는 NVIDIA GeForce 8800 GTX를 이용하여 AES를 최적화 하였다. GPU에서 구동된 AES는 CPU에서 연산을 수행한 경우보다 약 8배에서 20배까지 연산 속도가 증가 했다. CPU에서 동작하는 AES의 경우에는 9번의 라운드를 반복하는 동안 1 라운드에 1번씩 S-box와 Key 값을 참조하여 암호화를 수행 하므로 총 9번 참조해야 했다. 하지만, GPU로 구현할 경우 이 값들을 공유메모리에 입력하여, 1번만 참조하면 되기 때문에 연산속도를 높일 수 있었다.

2009년에는 준동형 해쉬 (Homomorphic Hash) 알고리즘을 GeForce GTX 280을 이용하여 최적화시키는 연구가 진행되었다[5]. 이 연구에서는 각각의 해시 값을 각 스레드에서 연산이 가능하도록 설정하였다. 그 결과 CPU에서 구현되었을 때보다 38배 연산 속도가 증가 하였다.

2010년에는 NVIDIA GeForce 9500GT를 이용하여 RSA 알고리즘을 최적화하는 연구가 발표되었다[6]. 이 연구에서는 분할한 각각의 스레드(thread)에서 RSA 알고리즘이 수행되도록 설정한 뒤, 평문을 동일한 크기로 나누어 각 스레드에 입력하여 암호화하도록 설정하였다. 이 경우 스레드의 개수와 평문의 크기에 따라 최소 2배정도의 연산시간이 감소하였다.

2011년에 진행된 연구에서는 MD5 해시 알고리즘을 GeForce GTX 9800을 이용하여 최적화 하였다[7]. 이 연구에서는 알고리즘을 병렬화하기 위해 for문과 같은 반복문을 제거하고, 반복횟수 만큼 스레드로 분할 하였다. 그리고 메시지 값들은 각 스레드에 저장하여 연산하였다. 그 결과, CPU에서 구현되는 MD5 해시 알고리즘보다 약 16배 정도의 연산속도가 증가하였다.

2013년에는 NVIDIA GTX 680을 활용하여 LEA를 최적화하였다[8]. 이 연구에서는 LEA 알고리즘을 스레드로 분할하였으며, 글로벌 메모리에서의 메모리 충돌을 방지하기 위해 CUDA의 Coalesced memory 기법을 활용하여 LEA를 최적화 시켰다. 그 결과 CPU에서 구현한 결과보다 약 2.8배의 연산속도 증가를 확인하였다.

위에서 언급한 연구외에도 GPU를 이용한 암호의 최적화 연구들이 진행되어 왔지만, 대부분의 연구는 영상처리에 중점을 둔 GPU를 기반으로하여 최적화가 진행되었다. 하지만 본 논문에서는 일반적인 연산을 처리하는데 효율성이 높은 GPGPU인 NVIDIA

Tesla K20으로 연구를 진행하였다. 또한 오프칩 메모리인 글로벌 메모리와 스레드 분할을 사용하여 LEA를 최적화한 앞선 연구와는 다르게 온칩 메모리(On-chip memory)인 공유메모리 (Shared memory)와 각 스레드에 Key 값을 분배 시, 발생 가능한 뱅크 충돌을 방지하는 구현 방법을 활용하여 LEA 최적화 연구를 진행하였다.

2.3 LEA 알고리즘

LEA (Lightweight Encryption Algorithm)는 2012년 국가보안기술연구소에서 개발한 암호이다[9]. LEA는 대용량 데이터의 고속 암호화가 가능하도록 설계되었다. LEA는 128bit 블록 단위로 암·복호화를 진행하는 블록 암호이다. LEA알고리즘은 모듈러 덧셈, XOR연산, bit를 이동하는 ARX 연산으로 구성되어 있다. 본 문단에서는 128bit LEA의 키스케줄링(Key scheduling)과 암호화(encryption) 알고리즘에 대하여 설명하고자한다.

우선 키스케줄링에 대해서 살펴보면 아래와 같다. 128bit LEA의 키스케줄링에서는 아래의 4개의 상수가 사용 된다.

$$\delta[0] = 0xc3efe9db \tag{1}$$

$$\delta[1] = 0x44626b02 \tag{2}$$

$$\delta[2] = 0x79e27c8a\delta \tag{3}$$

$$\delta[3] = 0x78df30ec \tag{4}$$

이때, 키 값은 K=(K[0],K[1],K[2],K[3])으로 설정하고. 라운드 키는 RK=(RKi[0], RKi[1],…,RKi[5]) (0<i<24)로 설정한다. 초기 키 값은 키 스케줄링에서 사용되는 내부상태 변수 T=(T[0],T[1],T[2],T[3])로 변환한다. ROLi(x)는 32비트 비트열 x의 i bit 좌측으로 순환이동을 의미하며, RORi(x)는 32비트 비트열 x의 i bit 우측으로 순환 이동시키는 것을 의미한다. 그리고 ⊞는 두 비트열의 덧셈에 모듈러 2^{32}를 한 연산기호이다.

$$T[1] \leftarrow ROL_3(T[1] \boxplus ROL_{i+1}(\delta[i \bmod 4])) \tag{5}$$

$$T[2] \leftarrow ROL_6(T[2] \boxplus ROL_{i+2}(\delta[i \bmod 4])) \tag{6}$$

$$T[3] \leftarrow ROL_{11}(T[3] \boxplus ROL_{i+3}(\delta[i \bmod 4])) \tag{7}$$

$$RK_i \leftarrow (T[0], T[1], T[2], T[1], T[3], T[1]) \tag{8}$$

키스케줄링 알고리즘은 위의 수식과 같이 모듈러 덧셈과 bit 이동을 통하여 총 24라운드를 반복하도록 구성되어 있다.

LEA의 암호화 알고리즘은 아래의 수식과 같다. 128bit LEA암호화 알고리즘은 총 24 라운드를 반복하여 평문을 암호화 한다. 이때, 평문은 바이트 배열 P=(P[0],P[1],P[2],P[3])로 구성되며, 초기 입력 값은 라운드함수에서 출력되는 내부 상태 변수 X_{i+1} = (X_{i+1}[0], …,X_{i+1}[3]) (0<i<23) 로 설정한다. 마지막 X_{24}는 암호문 C=(C[0],C[1],C[2],C[3])로 출력하게 된다. LEA 암호화 과정을 수식으로 표현하면 다음과 같다.

$$X_{i+1}[0] \leftarrow ROL9((X_i[0] \oplus RK_i[0]) \boxplus (X_i[1] \oplus RK_i[1])) \tag{9}$$

$$X_{i+1}[1] \leftarrow ROR_5((X_i[1] \oplus RK_i[2]) \boxplus (X_i[2] \oplus RK_i[3])) \tag{10}$$

$$X_{i+1}[2] \leftarrow ROR3((X_i[2] \oplus RK_i[4]) \boxplus (X_i[3] \oplus RK_i[5])) \tag{11}$$

$$X_{i+1}[3] \leftarrow X_i[0] \tag{12}$$

$$C[0] \leftarrow X_r[0],\ C[1] \leftarrow X_r[1],\ C[2] \leftarrow X_r[2],\ C[3] \leftarrow X_r[3] \tag{13}$$

LEA암호화 과정은 키스케줄링에서 연산된 라운드 키와 평문간의 모듈러연산과 비트 이동연산으로 하나의 라운드가 진행된다. 라운드가 총 24번 반복하여 평문이 암호화 된다. 아래의 Fig.1.은 i번째 암호화 과정을 도식화한 것이다[10].

III. CUDA를 이용한 LEA 최적화

본 단원에서는 CUDA를 이용하여 LEA를 최적화 하는 방법에 대하여 설명하고자 한다. 본 연구는 GPGPU인 NVIDIA Tesla K20을 사용한 환경에서 진행 되었다. LEA를 최적화하기 위해 스레드를 분할과 공유 메모리를 함께 사용한 방법을 이용해 LEA를 최적화 하였다.

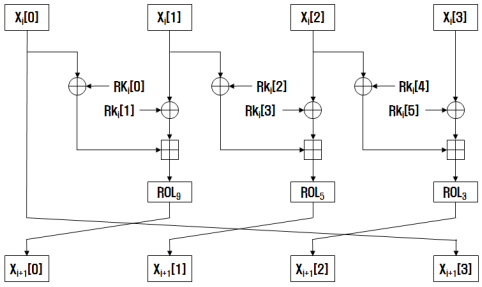

Fig. 5. LEA Algorithm at i th round

3.1 제안 방법

본 연구에서는 스레드 사용과 공유메모리 사용이라는 두가지 기법에 기반한 최적화된 모델을 제안한다. 우선 CUDA를 이용한 암호알고리즘들의 최적화 기법으로 일반적으로 이용되는 스레드의 분할을 이 논문에서도 이용한다. 우리는 이 논문에서 다중 스레드 기법을 사용하기 위해서 CUDA의 기본적인 SIMT (Single Instruction and Multiple Threads - 하나의 명령으로 다중 스레드 실행)를 이용하였다. 즉, Fig. 6.에서처럼 1차원으로 스레드를 분할하였고, 각 스레드에서 LEA 알고리즘이 수행하도록 설정하였다.

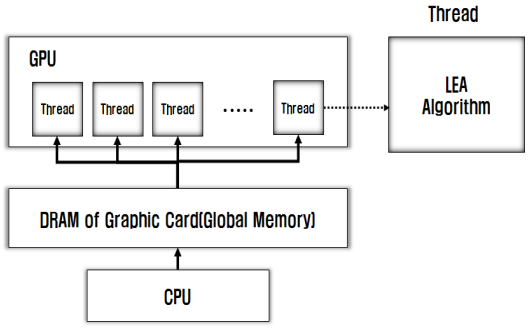

Fig. 6. Optimization of LEA Using Split Thread

본 연구는 위의 스레드 분할 뿐만 아니라 관련연구에서 설명한 CUDA의 메모리구조에서의 온칩과 오프칩 메모리 특징을 추가적으로 활용하였다.

본 연구는 [5]에서 진행한 연구와는 다르게 CUDA의 온칩 메모리에 있는 공유메모리를 이용하였다. 공유 메모리는 16KB의 용량을 동등한 속도로 L1캐시로 사용할 수 있다는 장점이 있다. 이러한 이유로 본 논문에서는 Fig. 7.과 같이 공유메모리에

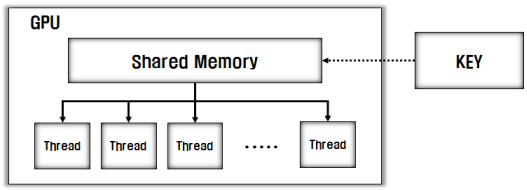

Fig. 7. Optimization of LEA Using Shared Memory

LEA 알고리즘에서 공통적으로 사용되는 키 (Key) 값을 저장하여 효율적으로 제어 및 관리를 하였다.

3.2 최적화 방안

본 연구에서는 GPGPU인 Tesla K20을 이용하여 최적화를 진행하였다. 공유 메모리로 최적화할 경우 2가지 사항을 고려해야 한다.

첫 번째로는 메모리를 구성하는 메모리 뱅크의 엑세스 충돌로 인해 발생하는 뱅크 충돌을 고려해야 한다. 본 연구에서는 Fig.8.과 같이 뱅크 충돌을 방지하기 위해 코드를 공유메모리에 구현하였다.

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
        __shared__ unsigned char s_key[32];
        s_key[tid]=dev_key[tid];
        __syncthreads();
unsigned char key;
key=s_key[tid];
```

Fig. 8. Code of Shared Memory in GPU

이 때, 뱅크를 도식화하면 Fig. 9.와 같이 16개의 뱅크로 나눌 수 있다.

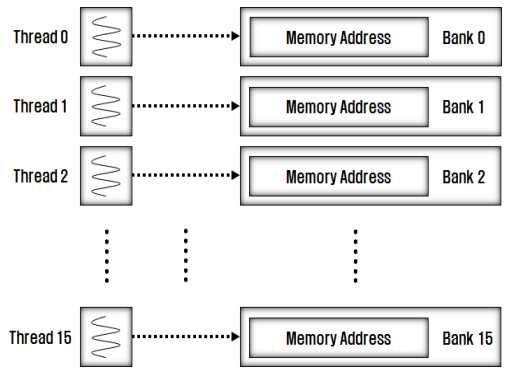

Fig. 9. Ideal Bank Conflict of Shared Memory

Fig.9.는 이상적으로 나누어진 뱅크 충돌을 표현한 그림이다. 그러나 각각 하나의 스레드가 서로 다른 뱅크에 접속하여 뱅크 충돌만 이루어지지 않는다면, Fig.10.에서 볼 수 있듯이 그 경우에도 성능이 향상될 수 있다.

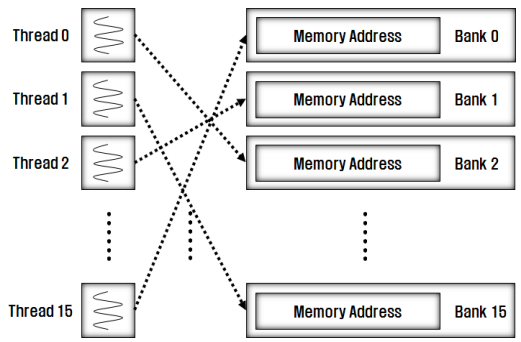

Fig. 10. General Bank Conflict of Shared Memory

두 번째 고려사항은 워프 현상이다. 워프 현상이란 커널의 코드가 메모리를 읽고 사용할 경우, 해당 작업을 수행하는 동안 다른 워프를 스위칭하여 연산을 수행하게 되는 방식이다. CUDA의 겨우 하나의 SM에서 프로세스를 진행할 때, 스레드의 개수가 32개 이상이라면 워프 현상이 발생하게 된다. CUDA 연산시 각 코드를 워프 단위로 동시에 처리하게 된다. 레지스터에 저장된 변수에 대한 연산은 하나의 워프 단위로 처리하고, 메모리에서 레지스터로 불러들이는 연산시, 워프 단위의 절반인 16개의 스레드로 나누어 진행한다. 그러나 분기문이 존재하는 코드의 경우 각 스레드별로 연산하면 그 효율이 감소한다.

위의 2가지 사항을 고려하여 앞서 제안한 방법을 구현하면 가장 최적화된 결과 값을 확인할 수 있다.

IV. 실험결과

본 연구에서는 스레드 분할 및 공유메모리를 활용하여 LEA의 연산속도를 높이는 연구를 진행 하였다. 이를 실험하기 위해 CPU는 Intel Xeon E5 2620, GPU는 Tesla K20을 이용하였다. 또한, GPU에서 구동되는 커널함수의 정확한 연산시간과 GPU 점유율을 측정할 수 있는 NVIDIA의 Nsight를 사용하였다[1]. 위에서 제안한 방법을 이용하여 1개의 블록에 16개, 32개, 128개, 256개, 512개의 스레드의 연산 시간을 측정한 결과는 다음

Table 1. Optimization of LEA Using Thread

Number of Threads	Computing Time(μs)	Occupancy rate of GPU
1	148.256	25%
16	110.752	25%
32	146.112	25%
64	108.544	50%
128	145.568	100%
256	112.032	100%
512	135.104	100%

의 Table 1.과 같다.

Table 1.에서 볼 수 있듯이, 하나의 블록에서 스레드를 분할하여 측정한 연구 결과, 스레드의 개수를 16개, 64개, 256개로 설정 하였을 경우, CPU에서 연산되는 LEA의 처리 시간(823.136 μs)보다, GPU로 처리할 경우, 약 7.5배 이상 감소하는 것을 확인 하였다. 하나의 블록에 대한 이러한 성능 결과와 함께 본 논문에서는 블록이 하나 이상일 때 어떠한 변화가 일어나는지를 조사하였다. Fig.11. 은 16개, 64개, 256개에 대해 블록 수를 변화하여 연산시간을 측정한 결과를 그래프로 표현한 것이다.

요약하자면, 하나의 블록에서는 64개의 스레드를 사용한 경우 최소의 연산 시간과 50%라는 최소의 GPU점유율을 갖는다. 하지만, 블록의 수가 증가할 경우, 64개의 스레드를 갖는 모델에서 메모리 점유율은 100%로 증가 하였고 대신에 하나의 블록을 사용할 때 성능이 떨어지던 256개의 스레드 사용 모델이 오히려 가장 짧은 연산시간 측정값을 갖는 최적의 모델이 됨을 알 수 있다.

이와 같은 결과가 측정되는 이유는 최적화 방안에

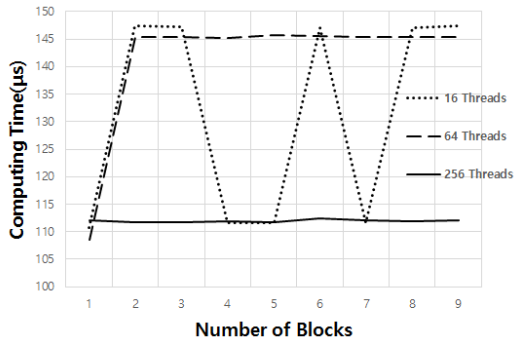

Fig. 11. Operation time change according to the number of blocks

서 언급한 워프 현상과 뱅크, 공유메모리가 할당 할 수 있는 스레드 개수 때문이다.

한 번의 워프에서 LEA 코드가 32개의 스레드에서 동작하고, 이 작업을 수행하는 동안 다른 워프를 스위칭하여 32개의 스레드에서 LEA코드를 수행하게 된다. 그러므로 32개의 스레드를 활용하고 스위칭 할 수 있는 여분의 32개의 스레드를 가질 수 있는 64개의 스레드를 설정 할 경우 가장 빠른 결과값을 얻었다. 또한, LEA 코드 내에는 분기문이 존재한다. 본 연구에서는 각 스레드별로 LEA 코드를 수행하도록 처리하였기 때문에 워프 현상이 발생할 경우, 그 속도가 감소하는 현상이 측정되었다.

16개의 스레드인 경우에는 워프 현상은 발생하지 않지만, 스레드가 엑세스 할 수 있는 공유메모리의 뱅크 수가 16개이기 때문에 연산시간이 감소하였다.

128개이상의 스레드를 설정할 경우 256개의 스레드에서 가장 빠른 결과 값을 얻었다. 그 이유는 공유메모리가 할당할 수 있는 스레드의 개수가 256개이기 때문이다. 각각의 스레드에서 LEA를 수행할 때, 필요한 키 값은 하나이다. 이 경우 각 스레드에서 연산 시 공유메모리의 키 값을 한번만 참조한다. 이때, 할 당 가능한 개수가 256개이므로 스레드를 256개 설정 하였을 때 가장 빠른 연산 결과를 얻었다.

그러나 1개, 128개의 스레드의 경우와 스레드 256개를 제외하고 블록의 수가 증가 할 때, 연산 시간이 감소한다. 그 이유는 메모리에서 레지스터로 데이터를 읽어오고 레지스터의 값을 메모리로 사용하는 연산으로 인해 연산시간이 느려지게 된다.

위의 연구 결과를 바탕으로 2013년 ICISC에서 발표된 CUDA LEA를 구현하여 Tesla K20으로 구동한 결과 [5]와 본 연구에서 하나의 블록에 64개의 스레드로 분할한 LEA와 성능을 비교 하면 약 2.5배로 연산시간이 빨라지는 것을 확인할 수 있었다. 그러나 LEA가 대용량 연산처리에 중점을 두어 개발되었다는 점에서 블록의 수가 증가하여도 연산시간에 크게 변동이 없는 256개의 스레드를 분할한 경우가 가장 최적화된 LEA 암호이다.

V. 결 론

본 논문에서는 LEA를 각각의 스레드에서 LEA 알고리즘을 수행하고 키 값을 공유메모리에 입력하는 방법으로 연산속도를 증가 시켰다. CPU에서 구동되는 LEA자체의 속도만으로도 충분히 빠른 연산 속도를 가졌지만, 대용량을 처리하는데 있어서 느려 질 수밖에 없다. 본 논문에서는 블록의 수에 관계없이 256개의 스레드를 설정할 경우 기존 LEA보다 7.5배 빠른 결과를 얻을 수 있었다. Tesla K20을 사용했음에도 불구하고 하나의 데이터를 암호화하는데 걸린 시간이 7.5배 감소하였다는 것은 완벽한 최적화 방안으로 볼 수 없을 수 있지만, LEA가 대용량 데이터 보안을 위한 암호 알고리즘이라는 점에서 대용량 데이터 처리시 연산시간은 추가적으로 감소 될 수 있다. 그 이유는 블록 수가 연산 속도에 크게 영향을 미치지 않고, 대용량을 처리를 256개의 스레드를 가지는 여러 개의 블록으로 병렬 처리할 수 있기 때문이다. 향후 연구에서는 CUDA를 활용하여 크롤링, 데이터 마이닝 등 연산속도를 증가 시킬 수 있는 분야에 대하여 연구를 진행할 예정이다.

References

[1] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture: Programming Guide (Version 7.0), http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3huHVgmRs, 2014.

[2] Jung, Yeoung Hun, CUDA Parallel Programming, FREELEC, 532-12, Sang 3-dong, Wonmi-gu, Bucheon-si, Gyeonggi-do, Korea, 326 .2011

[3] NVIDIA.http://kr.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf

[4] Manavski and Svetlin A. "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography." Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on. IEEE, pp. 65-68, 2007.

[5] Zhao, Kaiyong, et al. "Speeding up homomorpic hashing using GPUs." Communications, 2009. ICC'09. IEEE International Conference on. IEEE, pp. 1-5, 2009.

[6] Fan, Wenjun, Xudong Chen, and Xuefeng Li. "Parallelization of RSA algorithm based on compute unified device

architecture." Grid and Cooperative Computing (GCC), 2010 9th International Conference on. IEEE, pp.174-178. 2010.
[7] Wu, Hongwei, Liu, Xiangnan, and Tang, Weibin. "A fast GPU-based implementation for MD5 hash reverse." Anti-Counterfeiting, Security and Identification (ASID), 2011 IEEE International Conference on. IEEE, pp.13-16, 2011.
[8] Seo.Hwajeong, et al. "Parallel Implementations of LEA." Information Security and Cryptology—ICISC 2013. Springer International Publishing, pp.256-274, 2014.
[9] Hong.Deukjo, et al. "LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors." Information Security Applications. Springer International Publishing, pp.3-27, 2014.
[10] Hong.Deukjo, Kim.Dongchan and Kwon.Daeseong, 128-Bit Lightweight Block Cipher LEA, TTAK.KO-12.0223, Telecommunications Technology Association, Dec 2013.

〈저 자 소 개〉

박 무 규 (Moo Kyu Park) 학생회원
2014년 2월: 세종대학교 물리학과 학사 졸업
2014년 3월~현재: 고려대학교 정보보호학과 통합과정
〈관심분야〉 정보보호, 빅데이터 분석 기술

윤 지 원 (Ji Won Yoon) 종신회원
2003년 2월: 성균관 대학교 정보공학사 졸업
2005년 2월: University of Edinburgh, 정보학과 석사 졸업
2008년 11월: University of Cambridge 전자공학과 박사 졸업
2008년 2월~2009년 5월: University of Oxford 로봇연구소 박사후과정
2009년 5월~2011년 5월: University of Dublin 통계학과 연구원 및 강사
2011년 7월~2012년 8월: IBM 연구소 정규 연구원
2012년 9월~현재: 고려대학교 정보보호대학원 조교수
〈관심분야〉 신호정보처리, 응용통계, 빅데이터 분석 기술, 도감청 탐지 기술