Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.3, Jun. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.3.423

NVIDIA CUDA PTX를 활용한 SPECK, SIMON, SIMECK 병렬 구현*

장 경 배,[1†] 김 현 준,[1] 임 세 진,[2] 서 화 정[3‡]
[1,2,3]한성대학교 (대학원생, 학생, 교수)

Parallel Implementation of SPECK, SIMON and SIMECK by Using NVIDIA CUDA PTX*

Kyung-bae Jang,[1†] Hyun-jun Kim,[1] Se-jin Lim,[2] Hwa-jeong Seo[3‡]
[1,2,3]Hansung University (Graduate student, Undergraduate student, Professor)

요 약

SPECK과 SIMON은 NSA(National Security Agency)에서 개발한 경량블록암호이며 SIMECK은 SPECK과 SIMON의 장점을 결합하여 만든 새로운 경량블록암호이다. 본 논문에서는 SPECK, SIMON, SIMECK을 사용한 대용량 암호화를 구현 하는데 있어 병렬 처리에 용이한 GPU를 활용하였다. NVIDIA에서 제공하는 CUDA 라이브러리를 활용하였으며 불필요한 연산들을 제거하기 위해 CUDA 어셈블리 언어 PTX를 사용하여 성능을 극대화 하였다. 단순 CPU 구현과 GPU를 활용한 구현 결과를 비교해보았을 때, 더 빠른 속도로 대용량 암호화를 수행할 수 있었다. 또한 GPU 구현 시, C언어를 사용한 구현과 PTX를 사용한 구현을 비교해 보았을 때, PTX 사용 시, 성능이 더욱 증가하는 것을 확인하였다.

ABSTRACT

SPECK and SIMON are lightweight block ciphers developed by NSA(National Security Agency), and SIMECK is a new lightweight block cipher that combines the advantages of SPECK and SIMON. In this paper, a large-capacity encryption using SPECK, SIMON, and SIMECK is implemented using a GPU with efficient parallel processing. CUDA library provided by NVIDIA was used, and performance was maximized by using CUDA assembly language PTX to eliminate unnecessary operations. When comparing the results of the simple CPU implementation and the implementation using the GPU, it was possible to perform large-scale encryption at a faster speed. In addition, when comparing the implementation using the C language and the implementation using the PTX when implementing the GPU, it was confirmed that the performance increased further when using the PTX.

Keywords: SPECK, SIMON, SIMECK, CUDA, PTX

Received(02. 15. 2021), Modified(04. 28. 2021), Accepted(05. 11. 2021)
* 이 성과는 부분적으로 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.2018-0-00264, IoT 융합형 블록체인 플랫폼 보안 원천 기술 연구) 그리고 이 성과는 부분적으로 2021년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2020R1F1A1048478).
* 본 논문은 2020년도 한국정보보호학회 동계학술대회에서 발표한 우수논문을 개선 및 확장한 것임.
† 주저자, starj1023@gmail.com
‡ 교신저자, hwajeong84@gmail.com(Corresponding author)

I. 서 론

4차 산업혁명 시대에 진입함에 따라 단순 이미지 계산만이 아닌 다양한 분야에서 GPU가 활용 되고 있다 강력한 병렬 연산 능력을 보유한 GPU는 딥러닝, 블록체인 채굴, 시뮬레이션 등에서 필요한 연산을 CPU 보다 빠르게 처리할 수 있다 이러한 GPU의 계산력을 활용하여 프로그램을 만들 수 있도록 NVIDIA에서는 CUDA라는 병렬 프로그래밍 라이브러리를 제공한다. 현재까지 GPU를 활용하여 암호 알고리즘을 구현하는 연구가 활발히 진행되고 있다. Manavski는 AES 암호 알고리즘을 CUDA로 병렬 구현하였으며 CPU와 비교했을 때 5.92%의 성능을 향상 시켰다[1]. [2]는 ARIA, AES, DES 3가지 블록암호에 대해 CUDA를 활용하여 고속 구현하는 연구를 진행하였고 [3, 4]에서는 경량블록암호 LEA가 대용량 처리에 활용될 때, GPU를 활용하여 최적화 하였다. 이에 본 논문에서는 NVIDIA의 CUDA 라이브러리를 활용하여 경량블록암호 SIMON, SPECK[5], SIMECK[6]을 GPU 상에서 구현하였다. 이를 통해 대용량 병렬 암호화의 효율성을 확인한다. 또한 성능을 향상시키기 위하여 CUDA에서 제공하는 어셈블리 언어인 PTX(Parallel Thread Excution)로 모든 라운드 함수와 키 스케줄을 구현하였다. 본 논문의 구성은 다음과 같다. 2장에서는 GPU를 활용한 프로그래밍 기법인 CUDA 및 SPECK, SIMON, SIMECK의 주요 연산들에 대해 확인한다. 3장에서는 제안하는 CUDA PTX 구현 기법을 제시한다. 4장에서는 이에 대한 성능을 비교 분석하고 마지막으로 5장에서 본 논문의 결론을 내린다.

II. 관련 연구

2.1 CUDA

CUDA는 GPU의 프로그래밍을 위해 NVIDIA에서 개발한 병렬 컴퓨팅 라이브러리이다. GPU 가속 라이브러리, 컴파일러 등이 포함되어 있다. 개발자는 CUDA를 통해 GPU의 병렬 처리 연산을 활용하여 프로그램의 속도를 획기적으로 높일 수 있다. 전체적인 작업 흐름의 부분은 단일 스레드 성능에 최적화 된 CPU에서 실행되는 반면, 프로그램의 연산 집약적인 부분은 수천 개의 코어를 가지고 있는 GPU에서 병렬적으로 실행된다. 또한 C, C++, Python등 대중적인 프로그래밍 언어로 CUDA 프로그래밍이 가능하다.

2.2 CUDA 어셈블리 PTX

PTX는 CUDA 병렬 프로그래밍을 위한 명령어들의 집합이다. 어셈블리 언어와 같은 구조이며, 각 명령어는 수행할 연산 그리고 수행하는 연산의 대상들로 구성된다. CUDA에서 C와 같이 고급 언어로 작성된 코드는 컴파일 되어 PTX 명령어들을 생성한다. 이 과정에서 연산에 불필요한 명령어들이 생성될 수 있다. C 언어 구현과 비교하여, 하위 언어인 PTX로 프로그램을 구현한다면 컴파일 시 생성되는 불필요한 명령어들을 제외할 수 있어 GPU 상에서의 고속 구현이 가능하다.

2.3 경량블록암호 SPECK, SIMON, SIMECK

SIMON과 SPECK은 2013년 NSA에서 설계한 경량블록암호이다. SIMON은 하드웨어 구현에 최적화 되어있으며 SPECK은 소프트웨어 구현에 최적화 되어 있다. 2015년 CHES에서는 SIMON과 SPECK의 장점을 결합하여 새로운 블록암호를 SIMECK을 제안하였다. 라운드 함수는 SIMON의 라운드 함수를 개선하였으며 키 스케줄은 SPECK의 키 스케줄을 개선하였다. 그 결과, 안전성과 성능 두 가지 모두 향상 되었다. 세 가지 암호 모두, 키 스케줄을 통해 라운드 키들을 생성하고, 여러 번의 라운드 함수를 반복하여 평문을 암호화한다. 경량암호이므로 라운드 함수의 암호화 강도가 낮지만 여러 번 반복 수행함으로써 최종 암호문의 보안 강도를 높이는 구조를 가지고 있다.

2.4 SPECK

수식 이해를 위한 표기법은 Table.1.과 같다.

SPECK의 라운드 함수는 입력 값이 l_0, r_0 로 나누어지며 아래 수식을 수행한다.

$$\begin{aligned} l_i &= (ROR_7(l_i) \boxplus r_i) \oplus k_i \\ r_i &= ROL_3(r_i) \oplus l_i \end{aligned} \quad (1)$$

라운드 함수 구조는 Fig.1.과 같다.

Table 1. Notations

Notations	Meanings
l_0	Plaintext to be encrypted
r_0	Plaintext to be encrypted
k_i	Round key
$\oplus$	XOR operation
$\boxplus$	Modular addition operation
&	AND operation
ROL_i	Rotation left operation (i-bit)
ROR_i	Rotation right operation (i-bit)

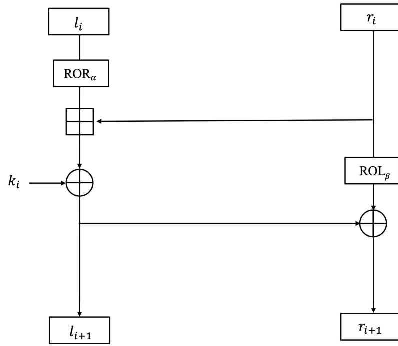

Fig. 1. Round function of SPECK

키 스케줄에서는 각 라운드 함수에서 사용 될 라운드 키 k_i를 생성한다. 초기 키워드 ($m=2,3$ 또는 4) $k=(k_0,l_0,\cdots,l_{m-2})$에 아래 키 스케줄을 수행하여 라운드 키를 생성한다. 블록 크기가 32-bit 인 경우 α와 β는 각각 7과 2이며, 나머지 블록 크기에 대한 α와 β는 각각 8과 3이다.

$$\begin{aligned} l_{i+m-1} &= k_i \boxplus ROR_\alpha(l_i) \oplus i \\ k_{i+1} &= ROL_\beta(k_i) \oplus l_{i+m-1} \end{aligned} \quad (2)$$

2.5 SIMON

SIMON의 라운드 함수는 입력 값이 l_0, r_0 로 나누어지며 아래 수식을 수행한다.

$$\begin{aligned} l_{i+1} &= f(l_i) \oplus ROL_2(l_i) \oplus r_i \oplus k_i \\ r_{i+1} &= l_i \\ f(x) &= ROL_1(x) \,\&\, ROL_8(x) \end{aligned} \quad (3)$$

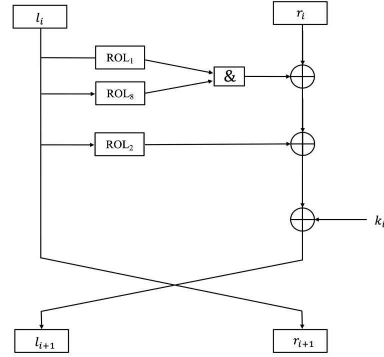

Fig. 2. Round function of SIMON

라운드 함수 구조는 Fig.2.와 같다.

SIMON은 초기 키워드 $m=2,3,4$에 따라 각각의 키 스케줄이 수행되는데 서로 매우 유사하기 때문에 본 장에서는 $m=4$인 경우에 대해서만 설명한다. 키 스케줄에서는 라운드 상수 C 그리고 시퀀스 값 $(z_j)_i$을 사용하며 수식은 아래와 같다.

$$\begin{aligned} k_{i+m} = c \oplus (z_j)_i \oplus k \oplus ROR_1(k_{i+1}) \oplus \\ ROR_4(k_{i+3}) \oplus ROR_3(k_{i+3}) \oplus k_{i+1} \end{aligned} \quad (4)$$

2.6 SIMECK

SIMECK은 Feistel 구조이며 라운드 함수는 SIMON의 라운드 함수와 유사하다. 라운드 함수의 입력 값이 l_0, r_0 로 나누어지며 아래 수식을 수행한다.

$$\begin{aligned} R_k(l_i, r_i) &= (r_i \oplus f(l_i) \oplus k_i, l_i)] \\ f(x) &= x \,\&\, ROL_5(x) \oplus ROL_1(x) \end{aligned} \quad (5)$$

라운드 함수 구조는 Fig.3.과 같다.

SIMECK의 키 스케줄은 SPECK의 키 스케줄과 유사하다. 초기 키워드는 (t_2, t_1, t_0, k_0)로 구성되며 첫 4 라운드 동안 사용된다. 다음 라운드부터 사용 될 라운드 키 k_i를 생성하기 위해 다음 수식의 키 스케줄이 수행된다. 키 스케줄에서는 라운드 함수 수식의 k_i 대신 라운드 상수 C 그리고 시퀀스 값 $(z_j)_i$을 XOR한 상수 값 $C\oplus(z_j)_i$ 을 사용한다.

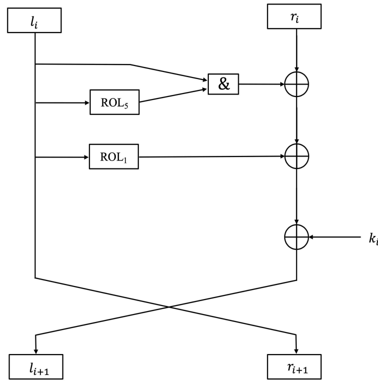

Fig. 3. Round function of SIMECK

$$k_{i+1} = t_i$$
$$t_{i+3} = k_i \oplus f(t_i) \oplus C \oplus (z_j)_i \quad (6)$$

III. 제안 기법

본 논문에서는 CUDA를 활용하여 GPU상에서 1024×15개, 1024×35개의 메시지를 SIMECK, SPECK, SIMON으로 병렬 암호화 하였으며 CPU 구현과의 성능을 비교하였다. GPU의 글로벌 메모리 접근 속도는 매우 느리기 때문에 CUDA 구현 시 메모리에 대한 잦은 접근은 성능을 감소시킨다. 따라서 사전에 모든 라운드 키를 생성하고 참조하는 것이 아닌 GPU에서 라운드 키를 생성하고 사용하는 방법을 활용하였다.

NIVIDIA GPU 및 CUDA 라이브러리를 활용하였으며 SIMON, SPECK, SIMECK 3가지 암호 알고리즘의 64-bit 평문, 128-bit 키 버전을 구현하였다. GPU는 다중의 블록과 스레드로 구성되어 있어 병렬처리에 용이하다. 본 논문에서는 다른 크기의 대용량 메시지인 1024×15개와 1024×35개의 메시지를 암호화하기 위해 15개, 35개의 블록을 사용, 블록 당 1024개의 스레드를 사용하였다. 각 스레드에 평문, 키 값을 할당하였고 하나의 스레드마다 한 번의 암호화를 수행하여 모든 메시지를 병렬로 암호화 할 수 있다. 제안하는 GPU 병렬 암호화 구조는 Fig.4.와 같다. 2가지 경우의 대용량 메시지 입력에 대해, 15개, 35개의 블록 모두 1024개의 메시지를 각각 할당받는다. 각 블록은 1024개의 스레드로 구성되어, 하나의 스레드마다 한번의 암호화를 수행한다.

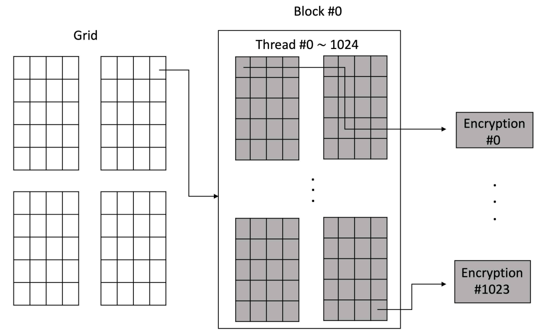

Fig. 4. Parallel encryption using GPU.

구현 성능을 최대로 높이기 위해 CUDA 어셈블리 언어인 PTX를 사용하였다. SIMON, SPECK, SIMECK의 라운드 함수와 키 스케줄 모두 PTX로 구현하여 컴파일 시 생기는 불필요한 연산을 제외하였다.

3.1 SPECK CUDA PTX 구현

SPECK의 라운드 함수, 키 스케줄 PTX 코드는 Fig.5, 6.과 같다. Fig.5.는 SPECK의 라운드 함수인 수식 1을 구현한 PTX 코드이다. 최종 암호문이 되기 위한 중간 값인 l_i, r_i 그리고 라운드 키 rk_i가 사용되며 로테이션 연산을 구현하기 위해 2개의 temp_result 변수를 사용하였다. CUDA PTX에서는 로테이션 연산 명령어를 따로 제공하지 않기 때

Algorithm : Round function (SPECK)

```
1: asm("{\n\t"
   // Rotation right (8-bit)
2:    "shr.b32 %1, %0, 8; \n\t"
3:    "shl.b32 %2, %0, 24; \n\t"
4:    "or.b32 %0, %1, %2; \n\t"
   // l_i = l_i + r_i (Modular addition)
5:    "add.u32 %0, %0, %3; \n\t"
   // r_i = r_i ⊕ rk_i (XOR-ing round key)
6:    "xor.b32 %0, %0, %4; \n\t"
   // Rotation left (3-bit)
7:    "shl.b32 %1, %3, 3; \n\t"
8:    "shr.b32 %2, %3, 29; \n\t"
9:    "or.b32 %3, %1, %2; \n\t"
      r_i = r_i ⊕ l_i
10:   "xor.b32 %3, %3, %0; \n\t"
      "}"
14: :  "+r"(l_i), "+r"(temp_result[0])
      "+r"(temp_result[1]), "+r"(r_i), "+r"(rk_i) );
```

Fig. 5. Round function PTX code (SPECK)

Algorithm : Keyschedule (SPECK)

```
1: asm("{\n\t"
  // rk[i] = A
2:   "mov.u32 %4, %0; \n\t"
  // Rotation right (8-bit)
3:   "shr.b32 %1, %3, 8; \n\t"
4:   "shl.b32 %2, %3, 24; \n\t"
5:   "or.b32 %3, %1, %2; \n\t"
  // B = B + A (Modular addition)
6:   "add.u32 %3, %3, %0; \n\t"
  // B = B⊕ i
7:   "xor.b32 %3, %3, %5; \n\t"
  // Rotation left (3-bit)
8:   "shl.b32 %1, %0, 3; \n\t"
9:   "shr.b32 %2, %0, 29; \n\t"
10:  "or.b32 %0, %1, %2; \n\t"
  // A = A ⊕ B
14:  "xor.b32 %0, %0, %3; \n\t"
  // i++
15:  "add.u32 %5, %5, 1; \n\t"
  // rk[i+1] = A
16:  "mov.u32 %8, %0; \n\t"
       "+r"(A), "+r"(temp_result[0]),
"+r"(temp_result[1]),
       "+r"(B), "+r"(rk[i]), "+r"(i), "+r"(rk[i+1]) );
```

Fig. 6. Keyschedule PTX code (SPECK)

문에 제공되는 시프트 연산과 or 연산 명령어를 활용하여 구현하였다. 32-bit 단위의 좌측 로테이션 연산은 i-bit 좌측 시프트 값과 (32-i)-bit 우측 시프트 값을 서로 OR 한 값과 같다. 우측 로테이션 연산은 i-bit 우측 시프트 값과 (32-i)-bit 좌측 시프트 값을 서로 OR 한 값과 같다.

SPECK-64/128에서 초기 키워드 크기는 4이다. 따라서 초기 키워드를 A, B, C, D라 정의하고 수식 2의 키 스케줄을 PTX 코드로 구현하였다. 제일 먼저 A, B가 입력되고 라운드 키는 A에 생성되어 라운드 키 $rk[i]$ 배열에 저장한다. i를 증가시키면서 하나의 입력 값은 A로 고정, 다른 입력 값은 B, C, D의 순서로 로테이션 하며 모든 라운드 키를 $rk[i]$에 채워 넣을 때 까지 반복한다. 참고로 6번 줄의 i를 증가시키는 명령어는 라운드 키인 $rk[i]$와 $rk[i+1]$의 배열 인덱스에 영향을 주지 않는다. Fig.6.은 A, B만을 입력 받은 키 스케줄 PTX 코드이다. 구현 코드에는 C, D도 입력 받아 같은 연산을 반복하지만 중복되기 때문에 본 논문에서는 생략하였다.

3.2 SIMON CUDA PTX 구현

SIMON의 라운드 함수, 키 스케줄 PTX 코드는 Fig.7, 8.과 같다.

Fig.7.의 2~8번 줄은 라운드 함수에서 $f(l_i)$ 함수에 해당한다. 32-bit 단위 $ROL_i(x)$ 연산은 2~4, 5~7번 줄에 해당하며 그 결과 $ROL_5(l_i)$, $ROL_1(l_i)$값이 result[0], [1]에 저장된다. 그리고 8, 9번 줄의 AND, XOR연산으로 $f(x)$ 함수를 마무리한다. 나머지 l_i의 로테이션 값과 라운드 키를 XOR하고 한 번의 라운드 함수가 종료된다. 그리고 바로 두 번째 라운드 함수를 첫 번째 라운드 함수의 연산과 동일하게 수행한다. SIMON의 라운드 함수는 l_i 와 r_i가 서로 바뀌는 특징이 있는데 라운드 함수 2번을 한 세트로 수행함으로 써 temp값에 mov 명령어로 l_i 또는 r_i를 저장하지 않고도 암호화를 수행할 수 있다. 두 번째 라운드 함수는 첫 번째 라운드 함수와 코드 구조가 동일하기 때문에 생략하였다.

SIMON 키 스케줄에는 이전 라운드 키들과 상수 c 그리고 $(z_j)_i$가 사용된다. 수식 4를 보면 알 수 있

Algorithm : Round function (SIMON)

```
1: asm("{\n\t"
   // First round function
   // Rotation left (1-bit)
2:   "shl.b32 %1, %0, 1; \n\t"
3:   "shr.b32 %2, %0, 31; \n\t"
4:   "or.b32 %3, %1, %2; \n\t"
   // Rotation left (8-bit)
5:   "shl.b32 %1, %0, 8; \n\t"
6:   "shr.b32 %2, %0, 24; \n\t"
7:   "or.b32 %4, %1, %2; \n\t"
   // f(x)
8:   "and.b32 %3, %3, %4; \n\t"
9:   "xor.b32 %5, %5, %3; \n\t"
  // Rotation left (2-bit)
10:  "shl.b32 %1, %0, 2; \n\t"
11:  "shr.b32 %2, %0, 30; \n\t"
12:  "or.b32 %1, %1, %2; \n\t"
13:  "xor.b32 %5, %5, %1; \n\t"
  // XOR-ing round key
14:  "xor.b32 %5, %5, %6; \n\t"
  // Second round function ...
15: :  "+r"(l_i), "+r"(temp_result[0]),
"+r"(temp_result[1]),
       "+r"(result[0]), "+r"(result[1]), "+r"(r_i),
"+r"(rk[i]),
       "+r"(rk[i+1]) );
```

Fig. 7. Round function PTX code (SIMON)

Algorithm : Keyshedule (SIMON)

```
1: asm("{\n\t"
    // XOR-ing c
2:   "xor.b32 %2, %0, %1; \n\t"
    // Rotation right (3-bit)
3:   "shr.b32 %4, %3, 3; \n\t"
4:   "shl.b32 %5, %3, 29; \n\t"
5:   "or.b32 %4, %4, %5; \n\t"
6:   "xor.b32 %2, %2, %4; \n\t"
7:   "xor.b32 %2, %2, %6; \n\t"
    // Rotation right (4-bit)
8:   "shr.b32 %4, %3, 4; \n\t"
9:   "shl.b32 %5, %3, 28; \n\t"
10:  "or.b32 %4, %4, %5; \n\t"
11:  "xor.b32 %2, %2, %4; \n\t"
    // Rotation right (1-bit)
12:  "shr.b32 %4, %6, 1; \n\t"
13:  "shl.b32 %5, %6, 31; \n\t"
14:  "or.b32 %4, %4, %5; \n\t"
15:  "xor.b32 %2, %2, %4; \n\t"
16: :  "+r"(c), "+r"(rk[i-4]), "+r"(rk[i]),
"+r"(rk[i-1]),
       "+r"(temp_result[0]), "+r"(temp_result[1]),
       "+r"(rk[i-3]) );
```

Fig. 8. Keyschedule PTX code (SIMON)

듯이, 로테이션 연산과 XOR 연산만이 사용된다. $(z_j)_i$ 값은 따로 연산 후 Fig.8.이 끝난 뒤, $rk[i]$에 XOR 하였다.

3.3 SIMECK CUDA PTX 구현

SIMECK의 라운드 함수, 키 스케줄 PTX 코드는 Fig.9, 10.과 같다.

라운드 함수의 입력 l_i는 출력 r_i가 되기 때문에 제일 먼저, 1번 줄의 mov 명령어로 l_i를 temp에 저장해둔다. Fig.9.의 2~10번 줄은 라운드 함수에서 f(l_i) 함수에 해당하며 앞서 SIMON과 매우 유사하다. $f(x)$ 값은 r_i 그리고 라운드 키 k_i와 XOR 하는데, 라운드 키는 항상 k_0에 저장된다. 이에 대해서는 키 스케줄 PTX 코드에서 확인할 수 있다. 마지막으로 입력 l_i 값이 저장 되었던 temp값을 mov 명령어로 출력 r_i에 저장하고 라운드 함수를 종료한다.

SIMECK의 키 스케줄은 라운드 함수 연산을 그대로 사용하기 때문에 코드에서의 차이점은 Fig.10. 14~17번 줄의 mov 명령어들이다. 라운드 함수에서 첫 번째 라운드 키 k_0가 사용되면 k_1이 두 번째

Algorithm : Round function (SIMECK)

```
1: asm("{\n\t"
2:      "mov.u32 %0, %1; \n\t"
   // Rotation left (5-bit)
3:      "shl.b32 %2, %1, 5; \n\t"
4:      "shr.b32 %3, %1, 27; \n\t"
5:      "or.b32 %4, %2, %3; \n\t"
   // Rotation left (1-bit)
6:      "shl.b32 %2, %1, 1; \n\t"
7:      "shr.b32 %3, %1, 31; \n\t"
8:      "or.b32 %5, %2, %3; \n\t"
   // f(x)
9:      "and.b32 %1, %1, %4; \n\t"
10:     "xor.b32 %1, %1, %5; \n\t"
   // r_i, k_0 XOR-ing
11:     "xor.b32 %1, %1, %6; \n\t"
12:     "xor.b32 %1, %1, %7; \n\t"
13:     "mov.u32 %6, %0; \n\t"
        "}"
14: :  "+r"(temp), "+r"(l_i), "+r"(temp_result[0])

       "+r"(temp_result[1]), "+r"(result[0]),
       "+r"(result[1]), "+r"(r_i), "+r"(k_0) );
```

Fig. 9. Round function PTX code (SIMECK)

Algorithm : Keyschedule (SIMECK)

```
1: asm("{\n\t"
2:      "mov.u32 %0, %1; \n\t"
   // Rotation left (5-bit)
3:      "shl.b32 %2, %1, 5; \n\t"
4      :"shr.b32 %3, %1, 27; \n\t"
5:      "or.b32 %4, %2, %3; \n\t"
   // Rotation left (1-bit)
6:      "shl.b32 %2, %1, 1; \n\t"
7:      "shr.b32 %3, %1, 31; \n\t"
8:      "or.b32 %5, %2, %3; \n\t"
   // f(x)
9:      "and.b32 %1, %1, %4; \n\t"
10:     "xor.b32 %1, %1, %5; \n\t"
   // k_0, Constant XOR-ing
11:     "xor.b32 %1, %1, %6; \n\t"
12:     "xor.b32 %1, %1, %7; \n\t"
13:     "mov.u32 %6, %0; \n\t"
14:     "mov.u32 %0, %1; \n\t"
15:     "mov.u32 %1, %8; \n\t"
16:     "mov.u32 %8, %9; \n\t"
17:     "mov.u32 %9, %0; \n\t"
        "}"
18: :  "+r"(temp), "+r"(k_1), "+r"(temp_result[0])
       "+r"(temp_result[1]), "+r"(result[0]),
"+r"(result[1]),
       "+r"(k_0), "+r"( C ⊕ (z_j)_i), "+r"(k_2),
"+r"(k_3) );
```

Fig. 10. Keyschedule PTX code (SIMECK)

라운드 키가 된다. 하지만 라운드 함수에서 항상 k_0를 라운드 키로 사용한다. 따라서 키 스케줄의 13번 줄은 temp에 저장한 k_1을 k_0로 옮겨 두 번째 라운드 키를 준비한다. 마찬가지로 15, 16번 줄은 기존 k_2,k_3를 k_1,k_2로 옮겨 세 번째, 네 번 째 라운드 키를 준비한다. 키 스케줄에서 생성한 라운드 키 값은 14, 17번 줄의 명령어를 통해 k_3에 저장되고 다섯 번 째 라운드 키로 사용된다.

IV. 성능평가

실험한 CPU, GPU 구현 환경은 Table.2.와 같다. 1024×15개와 1024×35개의 대량 메시지를 암호화할 때의 C언어를 활용한 CPU, GPU 구현 성능과 제안하는 PTX 코드를 사용한 GPU 구현의 성능은 Table.3.과 같다.

Table 2. Implementation environments

	CPU	GPU
Platform	Intel Core i7-10875H 2.3GHZ 8-core	RTX 2060
Programing language	C	C, PTX
Message encryption count	1024×15 1024×35	1024×15 1024×35

Table 3. Performance comparison (ms)

	Message	CPU (C)	GPU (C)	GPU (PTX)
SIMON 64/128	1024×15	6.5	0.45	0.4
SPECK 64/128		4.2	0.3	0.2
SIMECK 64/128		7.3	0.55	0.45
SIMON 64/128	1024×35	14.9	0.8	0.7
SPECK 64/128		9.5	0.5	0.4
SIMECK 64/128		15.4	1.0	0.9

C 언어를 사용한 CPU, GPU 구현 속도를 비교해 보았을 때 대용량 병렬 처리에 용이한 GPU에서 더욱 빠르게 수행할 수 있었다. 또한 GPU에서 C언어로 구현한 속도와 PTX로 구현한 속도를 비교해보았다. C 코드를 컴파일 하면 불필요한 명령어 셋이 생길 수 있다. 하지만 직접 작성한 PTX 코드는 사용된 명령어 셋들만을 사용하기 때문에 불필요한 연산들을 제외할 수 있다. PTX 코드와 C 코드의 성능을 비교해 본 결과, PTX 코드에서 SIMON은 12.5%, SPECK은 20%, SIMECK은 10%의 성능이 C 코드 대비 증가하였다.

V. 결 론

본 논문에서는 NVIDIA의 CUDA 라이브러리를 활용하여 SIMON, SPECK, SIMECK을 병렬로 암호화 하였다. 대용량 메시지를 암호화하는데 있어 GPU를 활용하였으며, CPU 대비 우수한 성능을 보여준다. 또한 CUDA 어셈블리 언어인 PTX로 암호화 연산들을 구현하여 GPU의 병렬 암호화 성능을 더욱 증가시켰다. 본 논문의 SPECK, SIMON, SIMECK의 PTX 코드는 Github[7]에 공개되어 있으며 GPU를 활용한 대용량 병렬 암호화에 사용될 수 있다.

References

[1] S. A. Manavski, ″CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography,″ Proceedings of the 2007 IEEE International Conference on Signal Processing and Communications, pp. 65-68, Nov. 2007.

[2] Y. Yeom, Y. Cho “High-Speed Implementations of Block Ciphers on Graphics Processing Units Using CUDA Library,” *Journal of the Korea Institute of Information Security & Cryptology*, 18(3), pp. 23-32, Jun. 2008.

[3] H. Seo, Z. Liu, T. Park, H. Kim, Y. Lee, J. Choi, and H. Kim, “Parallel im-

plementations of LEA," *In Information Security and Cryptology, ICISC 2013*, pp. 256-274, 2013.

[4] H. Seo, T. Park, S. Heo, G. Seo, B. Bae, Z. Hu, L. Zhou, Y.Nogami, Y. Zhu, H. Kim, "Parallel Implementations of LEA, revisted," *In Information Security Applications, WISA 2016*, pp. 318-330, 2016.

[5] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith and L. Wingers, "The SIMON and SPECK lightweight block ciphers," Proceedings of the 2015 IEEE Design Automation Conference, pp. 1-6, Jun. 2015.

[6] G. Yang, B. Zhu, V. Suder, M.D. Aagaard, , G. Gong, "The SIMECK family of lightweight block ciphers," *In Cryptographic Hardware and Embedded Systems, CHES 2015*, pp. 307 - 329. 2015.

[7] Github : source code [internet], https://github.com/starj1023/CUDA_PTX

〈저자소개〉

장 경 배 (Kyung-bae Jang) 학생회원
2019년 3월: 한성대학교 IT응용시스템공학부 졸업
2021년 3월: 한성대학교 IT융합공학부 석사
2021년 3월~현재: 한성대학교 정보컴퓨터공학과 박사과정
〈관심분야〉 정보보호, 암호, 양자컴퓨터

김 현 준 (Hyun-jun Kim) 학생회원
2019년 3월: 한성대학교 IT응용시스템공학부 졸업
2021년 3월: 한성대학교 IT융합공학부 석사
2021년 3월~현재: 한성대학교 정보컴퓨터공학과 박사과정
〈관심분야〉 부채널분석, 암호

임 세 진 (Se-jin Lim) 학생회원
2018년 3월~현재: 한성대학교 컴퓨터공학부 학사과정
〈관심분야〉 인공지능, 정보보호

서 화 정 (Hwa-jeong Seo) 종신회원
2010년 3월: 부산대학교 컴퓨터공학과 졸업
2012년 3월: 부산대학교 컴퓨터공학과 석사
2016년 3월: 부산대학교 컴퓨터공학과 박사
2016년~2017년: 싱가포르 과학기술청 연구원
2019년~현재: 한성대학교 IT융합공학부 조교수
〈관심분야〉 정보보호, 암호화 구현, IoT