Journal of The Korea Institute of Information Security & Cryptology
VOL.32, NO.6, Dec. 2022
ISSN 1598-3986(Print)
ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2022.32.6.1035

CUDA GPGPU 상에서 경량 블록 암호 PIPO의 최적 구현*

김 현 준,[1†] 엄 시 우,[1] 서 화 정[2‡]
[1,2]한성대학교 (대학원생, 교수)

Optimal Implementation of Lightweight Block Cipher PIPO on CUDA GPGPU*

Hyun-Jun Kim,[1†] Si-Woo Eum,[1] Hwa-Jeong Seo[2‡]
[1,2]Hansung University (postgraduate student, professor)

요 약

사물인터넷(IoT), 클라우드 컴퓨팅, 빅데이터 등의 확산으로 애플리케이션에 대한 고속 암호화의 필요성이 대두되고 있다. GPU 최적화는 GPU가 이론적으로 얻은 암호 분석 결과 또는 축소된 버전을 합리적인 시간에 검증하는데 사용될 수 있다. 본 논문에서는 다양한 환경에서 구현되고 있는 PIPO 경량암호를 대상으로 GPU 상에서 구현하였다. PIPO에 대한 무차별 대입 공격을 고려하여 최적 구현하였다. 특히 비트 슬라이싱 기법을 적용한 최적화 구현과 GPU 요소를 최대한 사용하였다. 결과적으로 제안 기법의 구현은 RTX 3060 환경에서 초당 약 195억의 처리량을 보여 이전 연구 보다 약 122배 높은 처리량을 달성하였다.

ABSTRACT

With the spread of the Internet of Things (IoT), cloud computing, and big data, the need for high-speed encryption for applications is emerging. GPU optimization can be used to validate cryptographic analysis results or reduced versions theoretically obtained by the GPU in a reasonable time. In this paper, PIPO lightweight encryption implemented in various environments was implemented on GPU. Optimally implemented considering the brute force attack on PIPO. In particular, the optimization implementation applying the bit slicing technique and the GPU elements were used as much as possible. As a result, the implementation of the proposed method showed a throughput of about 19.5 billion per second in the RTX 3060 environment, achieving a throughput of about 122 times higher than that of the previous study.

Keywords: PIPO, GPU, CUDA Implementation, Exclusive Key search

I. 서 론

사물인터넷, 클라우드 컴퓨팅, 빅데이터 등의 확산으로 애플리케이션에 대한 고속 암호화의 필요성이 대두되고 있다. 최신 그래픽 프로세서는 높은 처리능력을 제공하며, GPU(Graphics Processing Unit)의 획기적인 성능을 통해 암호 처리에 그래픽 프로세서를 활용하는 연구가 활발히 진행되고 있다

Received(08. 22. 2022), Modified(11. 08. 2022), Accepted(11. 08. 2022)
* 본 논문은 2022년 한국정보보호학회 하계학술대회에 발표한 우수 논문을 개선 및 확장한 것임.
* 본 연구는 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2021-0-00540, GPU/ASIC 기반 암호알고리즘 고속화 설계 및 구현 기술개발, 100%).
† 주저자, khj930704@gmail.com
‡ 교신저자, hwajeong84@gmail.com(Corresponding author)

[1-4]. GPU는 파일 또는 전체 디스크 암호화를 위한 암호화 보조 프로세서로 사용되어 CPU(Central Processing Unit) 암호화로 인한 성능 손실을 제거할 수 있다. 또한 CPU 사용량이 많은 SSL(Secure Sockets Layer) 서버에서 GPU를 보조 프로세서로 사용하면 암호화 부담에서 벗어나 다른 작업에 CPU 성능을 사용할 수 있다. 그리고 GPU 최적화는 GPU가 이론적으로 얻은 암호 분석 결과 또는 축소된 버전을 합리적인 시간에 검증하는 데 사용될 수 있다. 본 논문에서는 GPU 상에서 경량 블록 암호 PIPO[1]를 최적 구현하였다. PIPO는 ICISC 2020에서 처음 발표되었으며 제한된 리소스를 갖는 IoT 환경에 효율적이며 적은 비용으로 부채널 대응 기법을 적용할 수 있는 장점을 가진 경량 블록 암호이다. 다양한 환경에서 PIPO 구현에 대한 연구가 진행되어 왔다. Kim 등[2]은 CTR모드 최적화 기법을 사용하여 8비트 Atmega128과 x86 PC환경에서 PIPO64/128를 기존 레퍼런스 코드보다 각각 3.96%, 5.48% 성능 향상 시켰다. Eum 등[3]은 64비트 ARM 프로세서 환경에서 8개의 평문 블록과 16개의 평문 블록을 병렬 구현하였다. 벡터 레지스터와 벡터 명령어을 활용하여 기존 레퍼런스 코드 보다 각각 약 70%, 80%의 성능을 향상시켰다. Kwak 등[4]은 32비트 RISC-V 프로세서에서 PIPO 암호를 최적 구현 하였다. 레지스터 스케줄링, 32비트 레지스터 세트를 사용한 데이터 병렬 처리 접근, 내부 프로세스 결합 등의 최적화를 사용하여 PIPO-64/128를 단순 이식 버전보다 보다 128.5% 향상된 성능을 보였으며 참조 코드보다 393.52% 성능을 개선하였다. Jang 등[5]은 Grover의 알고리즘을 활용한 무차별 대입 공격을 위해 최적화된 양자 회로를 제시하였다. 또한 양자 시뮬레이터 ProejctQ에서 PIPO 블록 암호에 대한 Grover 검색 알고리즘의 양자 자원을 평가하였다. Lim 등[6]은 자바스크립트(Javascript), 웹어셈블리(WebAssembly)와 같은 웹 기반 언어를 사용하여 PIPO 64/128비트, 64/256비트를 구현하여 다양한 웹 브라우저와 OS 환경에서 성능평가를 수행하였다. Kwon 등[7]은 GPU상에서 PIPO 64/128를 병렬 구현하였다. Coarse grain 방식으로 구현하여 RTX 3070상에서 초당 약 1.59억 개의 암호 처리량을 달성하였다.

본 논문은 무차별 대입 공격을 고려하여 경량암호 PIPO를 CUDA GPU 환경에서 구현한다. 특히 높은 처리량을 위해 이전 연구에서 다루지 않은 고도의 비트 슬라이싱 기법을 사용한다. 비트 슬라이싱은 각 비트를 여러 블록으로 모아서 비트 단위로 연산한다. 여러 블록을 병렬로 처리되어 명령어 수를 줄일 수 있다. 다수의 블록(2개 이상의 블록)을 처리하는 고도의 비트 슬라이싱은 중간 결과를 저장할 다수의 레지스터 필요하다. GPU 장치는 많은 레지스터를 갖기 때문에 고도의 비트 슬라이싱에 적합하다. 구현에는 32개의 블록을 병렬로 연산하는 고도의 비트 슬라이싱 기법을 적용한다. 비트 슬라이싱 기법을 적용할 때 적용할 수 있는 기법인 S-BOX와 순열 연산에서의 비트 이동을 최적화와 그리고 비트 슬라이싱 방식의 카운터 기반 키 탐색을 함께 제안한다. 또한 GPU 최적 구현기법인 공유메모리 사용과 블록 당 스레드, 그리드 당 블록 수 조정, 루프 언롤링 기법을 적용하였다. 결과적으로 제안 기법의 구현은 RTX 3060 환경에서 초당 약 195억의 처리량을 보여 이전 연구 보다 약 122배 높은 처리량을 달성하였다.

II. 관련 연구

2.1 경량 블록 암호 PIPO

PIPO는 "Plug-In" 및 "Plug-Out"의 약자로, 각각 측면 채널 보호 및 비보호 환경에서의 사용을 말한다. IoT 환경에서 제한된 자원를 고려하여 설계되었으며 비선형 연산의 수를 최소화하는 데 중점을 둔 암호이다. 이러한 특징으로 PIPO는 부채널 분석 대응기법인 마스킹 기법을 효율적으로 적용할 수 있다. PIPO는 8 비트 AVR 구현 측면에서 128 비트 키를 사용하는 다른 경량 64 비트 블록 암호보다 성능이 뛰어나다. PIPO 암호의 구조는 Fig. 1. 과 같이 상태에 대해 다음 세 가지 기본 작업을 반복하는 구조로 구성된다. 첫 단계로 상태에 라운드 키의 64비트 워드를 추가하고 라운드 상수가 함께 추가된다. S-Layer는 8개의 8바이트 블록으로 나뉜 상태에서 각 블록의 비트를 8비트 S-box를 사용하여 치환한다. R-Layer는 상태에 8바이트 단위의 순환 연산이 적용된다. PIPO의 키 스케줄은 간단한 구조로 되어있다. PIPO-64/128의 경우 128비트의 마스터 키 K를 두 개의 64비트 하위 키 K0 및 K1으로 분할된다. 그런 다음 라운드 키는 $RK_i = K_{i \bmod 2}$ 이며 i는 $i = 0, 1, 2, 3, \cdots, 13$ 이다.

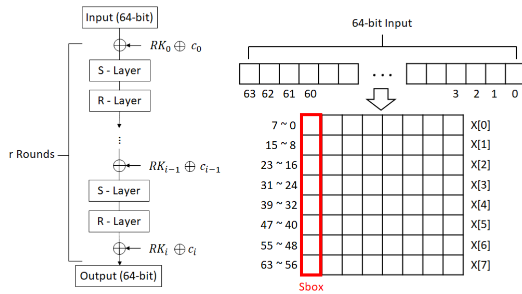

Fig. 1. Structure of PIPO block cipher[1]

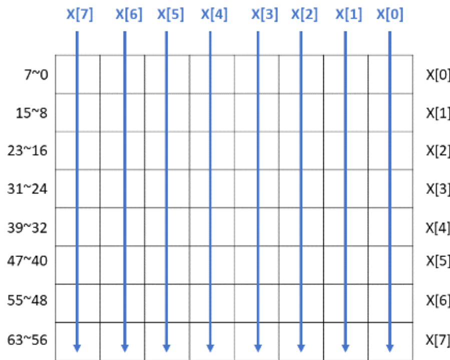

Fig. 2. Block structure after S-layer

PIPO-64/256의 경우 마스터 키 K는 4개의 64비트로 나뉜다.

2.2 비트 슬라이싱

비트 슬라이싱은 Biham[8]이 DES의 소프트웨어 구현 속도를 높이기 위해 조회 테이블 대신 처음 사용한 기술이다. 비트 슬라이싱은 여러 블록의 비트들을 각 순서대로 모아 비트 단위의 연산하는 기법을 말한다. 이러한 방식으로 여러 블록을 비트 연산 명령어와 병렬로 처리할 수 있다. 비트 슬라이싱 기법을 적용하기 위해서는 암호 알고리즘의 동작 과정을 AND, OR, NOT 등과 같이 간단한 논리 게이트의 조합으로 변경하여야 한다. 여러 블록 비트 슬라이싱 표현을 바꾸기 위한 packing 과정과 다시 원래 형태의 블록으로 되돌리기 위한 unpacking 과정이 추가되어 이러한 오버헤드도 함께 고려해야 한다.

2.3 CUDA 프로그래밍

CUDA(Compute Unified Device Architecture)는 GPU에서 수행하는 병렬 처리 알고리즘을 C 프로그래밍 언어를 비롯한 산업 표준 언어를 사용하여 작성할 수 있도록 하는 GPGPU 기술이다. CUDA는 엔비디아가 개발해오고 있으며 이 아키텍처를 사용하려면 엔비디아 GPU와 특별한 스트림 처리 드라이버가 필요하다. CUDA GPU 구조는 Fig. 3. 과 같이 GPU에서 실행되는 스레드 그룹 블록, 블록의 그룹 그리드, 32개의 스레드 묶음 워프가 있다. SM(Streaming Multi-processor)은 하나의 워프에서 스레드를 동시에 실행한다. GPU에서 병렬 실행되는 명령의 모음을 커널이라고 한다.

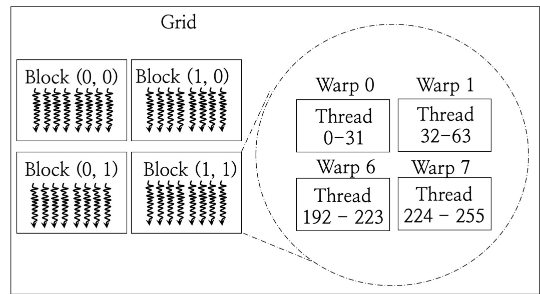

Fig. 3. Grid, Thread, Block and Warp configuration in CUDA

암호해독을 위해서는 하나의 스레드에서 하나의 패스워드를 해독하며 여러 개의 암호해독이 병렬로 이루어진다. 성능을 위해 블록당 스레드 수, 그리드당 블록 수의 적절한 값 선택 필요하며 데이터 전송 지연을 고려하여 충분한 스레드와 블록 사용이 필요하다. CUDA의 큰 장점으로 스레드 간에 공유할 수 있는 빠른 공유 메모리 영역에 접근할 수 있다. 제안 기법에서는 공유 메모리를 사용하여 높은 성능 향상을 보일 수 있었다.

III. 제안 기법

3.1 PIPO 비트 슬라이싱

비트 슬라이싱 기술은 여러 블록을 암호화하므로 GPU 환경과 같이 높은 처리량을 요구하는 환경에 적합하다. 병렬로 많은 블록을 계산하는 비트 슬라이싱 구현은 여러 연산 영역을 절약할 수 있지만 많은 레지스터를 필요하므로 오버헤드를 유발할 수 있다. GPU 환경은 이러한 고도의 비트 슬라이스 구현을 효율적으로 수행하기에 비교적 충분한 레지스터를 제공한다. GPU 코어의 32비트 연산 단위에 따라 64

비트 평문 32블록을 병렬 연산을 구현하였다. GPU가 수행하기에 충분한 레지스터를 제공하기 때문에 가능하였다. 비트 슬라이싱 기법 적용 시 32개의 평문이 병렬로 암호화되기 때문에 라운드 키도 32배 증가한다. 이러한 경우 커널은 글로벌 메모리에서 라운드 키를 불러오는 추가로 지연이 발생할 수 있다. 또한 지원하는 레지스터보다 많은 레지스터 사용으로 동작하지 않을 수 있다. 이러한 오버헤드를 방지하기 위해 제안하는 기법에서는 라운드 키를 사전에 생성하지 않고 on-the-fly 방식으로 계산한다. 라운드 키를 생성하는 추가 작업이 발생하지만, 일반적으로 경량암호의 라운드 키 생성은 복잡하지 않기 때문에 이러한 방식이 더 높은 성능을 보였다. Fig. 4. 는 제안기법의 동작 알고리즘이다. 스레드에서는 평문과 암호문을 입력받아 키를 생성하고 암호화 후 출력된 암호문과 실제 암호문을 비교한다. 이때 평문과 암호문은 패킹 상태로 입력받는다. 평문과 암호문은 키 탐색 과정에서 모두 같은 값을 사용하므로 한 번의 패킹 이후에 GPU에 전송된다. 과정의 첫 단계는 KeyGen에서 32개의 키를 생성한다. 그리고 한 라운드에서 AddRoundKey, S-Lyaer', R-Layer', KeyUpdate가 동작한다. AddRoundKey는 상태와 키가 XOR한다. 그리고 S-Layer', R-Lyaer'는 기존 S-Layer, R-Lyaer에 제안하는 비트 슬라이싱 기법이 적용된 함수이다. 그리고 KeyUpdate는 다음 라운드 키 생성한다. 라운드 반복 후 Verify 단계에서 실제 암호문과 비교하여 동일한 경우 키 값을 반환한다. 본 장에서는 효율적인 KeyGen과 최적화된 비트 슬라이싱 기반의 라운드 함수 구현 기법을 제안한다.

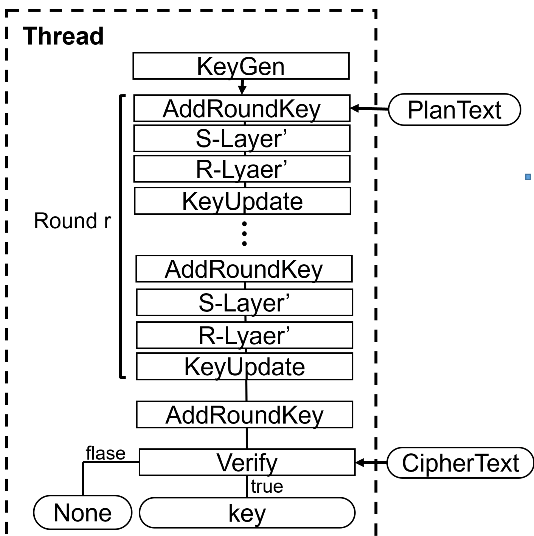

Fig. 4. The process of the proposed technique

3.2 S-Layer

기존 S-Layer에서는 Fig. 2. 와 같이 바이트 단위의 비트 슬라이싱 연산이 적용된다. 즉 8개의 평문 바이트는 비트 단위로 묶여 S-box에 입력된다. 이를 반영하여 제안기법의 GPU 상에서 구현은 Fig. 5. 와 같이 S-Box에 비트가 입력된다. 여기서 배열 X는 비트 슬라이싱 표현으로 32개의 64비트 평문이다. 따라서 X[0]은 평문에는 모든 평문의 0번째 비트, X[1]은 평문에는 64비트 평문의 1번째 비트가 저장된다. PIPO는 8비트 프로세서에서 효율적인 구현을 목표로 설계되어 R-Layer에서 8비트 로테이션 연산만으로 구성된다. 이를 위해서 S-Box의 마지막 연산에서는 비트 교환이 이루어진다. 제안기법에서는 비트 교환을 R-Layer에서 수행하여 명령어 수를 줄인다. 사용된 S-Box는 Fig. 6.과 같다.

```
1. __device__ void pipo_sBoxLayer(uint32_t* X) {
2. register uint32_t T0, T1, T2;
3. PIPO_SBOX(X[0], X[8], X[16], X[24], X[32],
   X[40], X[48], X[56]);
4. PIPO_SBOX(X[1], X[9], X[17], X[25], X[33],
   X[41], X[49], X[57]);
5. PIPO_SBOX(X[2], X[10], X[18], X[26], X[34],
   X[42], X[50], X[58]);
6. PIPO_SBOX(X[3], X[11], X[19], X[27], X[35],
   X[43], X[51], X[59]);
7. PIPO_SBOX(X[4], X[12], X[20], X[28], X[36],
   X[44], X[52], X[60]);
8. PIPO_SBOX(X[5], X[13], X[21], X[29], X[37],
   X[45], X[53], X[61]);
9. PIPO_SBOX(X[6], X[14], X[22], X[30], X[38],
   X[46], X[54], X[62]);
10. PIPO_SBOX(X[7], X[15], X[23], X[31], X[39],
    X[47], X[55], X[63]);
11. }
```

Fig. 5. Implementation of s-Layer with bit slicing on GPU

```
1. #define PIPO_SBOX(X0, X1, X2, X3, X4, X5, X6, X7)\
2. X5 ^= (X7 & X6); X4 ^= (X3 & X5); \
3. X7 ^= X4; X6 ^= X3; \
4. X3 ^= (X4 | X5); X5 ^= X7; \
5. X4 ^= (X5 & X6); X2 ^= X1 & X0; \
6. X0 ^= X2 | X1; X1 ^= X2 | X0; \
7. X2 = ~X2; X7 ^= X1; \
8. X3 ^= X2; X4 ^= X0; \
9. X6 ^= (X7 & X5); T0 = X7^X6;\
10. X6 ^= (X4 | X3); T1 = X3^X5          \
11. X5 ^= (X6 | X4); X2 ^= T0;           \
12. T2 = X7; X1 = X1 ^ X4 ^ (T1 & T0);\X0 = X0^T1; \
```

Fig. 6. S-Box implementation of the proposed method

3.3 R-Layer

제안 기법에서는 32블록을 병렬로 수행하는 비트 슬라이싱을 적용하였다. 따라서 비트의 교환은 레지스터 간에 값 교환으로 연산할 수 있다. 따라서 R-Layer의 8비트 로테이션은 레지스터 간 값 교환으로 처리된다. 이러한 점으로 S-Box에서 비트 교환을 하지 않고 R-Layer에서 수행하면 연산량을 줄일 수 있다. Fig. 7. 은 제안기법을 적용한 R-Layer 구현이다. 기존의 로테이션 연산은 값 교환으로 처리되며 S-Layer에서 생략된 비트 교환은 인덱스 변경으로 간단하게 수행될 수 있다. 결과적으로 S-Layer에서 비트 교환을 위한 64번의 연산이 줄어들고 R-Layer에서 24번의 연산이 늘어난다.

3.4 카운터 기반 키 탐색

많은 양의 데이터를 암호화하기 위해서는 암호화를 위해 CPU에서 GPU로 평문을 전송해야 하므로 추가적인 지연이 발생한다. Tezcan[9]은 카운터값을 커널에서 키 값으로 생성하고 스레드마다 하나씩 증가하는 카운터 모드를 사용하여 일반 텍스트를 GPU로 전송하는 프로세스를 제거하였다. 이 아이디어를 채택하여 비트 슬라이싱 기술에 적용하였다. 그러나 일반 표현의 카운터 값을 비트 슬라이싱 표현으로 변환하면 오버헤드가 발생한다. 따라서 제안기법에서는 Fig. 8. 10-12와 같이 이미 비트 슬라이싱 표현에 있는 카운터를 생성하여 변환 과정을 생략한다. 제안하는 방법은 카운터 값을 생성한 후 비트 슬라이싱 표현식으로 패킹하지 않고 패킹된 상태로 카운터를 생성한다. 32비트 블록은 동시에 처리된다. 따라서 0~4의 하위 키에는 같은 값을 사용하고 카운터는 32씩 증가된다. 다음 단계에서 스레드는 자신의 스레드 번호의 32배만큼 증가한 카운터 값을 키 값으로 사용한다.

```
1. __device__ void pipo_pLayer(uint32_t* X) {
2. uint32_t T;
3. T = X[15];
4. X[15] = X[8];
5. X[8] = X[9];
6. X[9] = X[10];
7. X[10] = X[11];
8. X[11] = X[12];
9. X[12] = X[13];
10. X[13] = X[14];
11. X[14] = T;
12. T = X[16];
13. X[16] = X[20];
14. X[20] = T;
15. T = X[17];
16. X[17] = X[21];
17. X[21] = T;
18. T = X[18];
19. X[18] = X[22];
20. X[22] = T;
21. T = X[19];
22. X[19] = X[23];
23. X[23] = T;
24. T = X[31];
25. X[31] = X[28];
26. X[28] = X[25];
27. X[25] = X[30];
28. X[30] = X[27];
29. X[27] = X[24];
30. X[24] = X[29];
31. X[29] = X[26];
32. X[26] = T;
33. T = X[39];
34. X[39] = X[33];
35. X[33] = X[35];
```

```
1. X[35] = X[37];
2. X[37] = T;
3. T = X[38];
4. X[38] = X[32];
5. X[32] = X[34];
6. X[34] = X[36];
7. X[36] = T;
8. T = X[47];
9. X[47] = X[42];
10. X[42] = X[45];
11. X[45] = X[40];
12. X[40] = X[43];
13. X[43] = X[46];
14. X[46] = X[41];
15. X[41] = X[44];
16. X[44] = T;
17. T = X[55];
18. X[55] = X[54];
19. X[54] = X[53];
20. X[53] = X[52];
21. X[52] = X[51];
22. X[51] = X[50];
23. X[50] = X[49];
24. X[49] = X[48];
25. X[48] = T;
26. T = X[63];
27. X[63] = X[61];
28. X[61] = X[59];
29. X[59] = X[57];
30. X[57] = T;
31. T = X[62];
32. X[62] = X[60];
33. X[60] = X[58];
34. X[58] = X[56];
35. X[56] = T;
36. }
```

Fig. 7. R-Layer implementation of the proposed method

```
1. uint32_t tid = blockIdx.x * blockDim.x + threadIdx.x;
2. uint32_t X[64];
3. uint32_t keys[128] = { 0, };
4. keys[0] = 0x55555555;
5. keys[1] = 0x33333333;
6. keys[2] = 0x0F0F0F0F;
7. keys[3] = 0x00FF00FF;
8. keys[4] = 0x0000FFFF;
9.
10. for (uint32_t i = 0; i < 128 - 5; i++) {
11.     keys[i + 5] = ((tid & (1 << i)) >> i) * 0xFFFFFFFF;
12. }
13.
14. uint32_t subkeys[64];
15. uint32_t subkeys2[64];
16.
17. for (int i = 0; i < 64; i++) {
18.     subkeys[i] = keys[i];
19. }
20.
21. for (int i = 0; i < 64; i++) {
22.     subkeys2[i] = keys[i + 64];
23. }
24.
25. for (int i = 0; i < 64; i++) {
26.     X[i] = plaintext[i];
27. }
```

Fig. 8. Counter-based exclusive key search on PIPO

3.5 공유 메모리 사용

CUDA 구현에서 메모리에서 자주 호출되는 값은 성능 향상을 위해 공유 메모리 영역에 저장한다. 제안 기법에서는 라운드 키 값을 공유 메모리 영역에 저장하여 높은 처리량을 달성 하였다. 하나의 암호 알고리즘이 동작할 때 키 값을 저장하기 위해 512바이트 필요하다. 각각의 스레드마다 하나의 암호 알고리즘이 동작하므로 키 값을 저장할 512바이트가 필요하다. 따라서 Fig. 9. 과 같이 라운드 키를 블록당 스레드 수만큼 곱한 만큼 공유 메모리를 사용한다. 이때 커널에서 사용할 수 있는 공유 메모리의 크기를 넘지 않도록 블록당 스레드의 수를 조절해야 한다. 블록당 스레드 수와 그리드 당 블록 수는 성능에 영향을 미치기 때문에 최적 성능을 달성하기 위해서는 적절한 선택 필요하다.

```
1. __shared__ uint32_t subkeys[64 * threadSize];
2. __shared__ uint32_t subkeys2[64 * threadSize];
3.
4. for (int i = 0; i < 64; i++)
5.   subkeys[i + threadIdx.x * 64] = key[i + tid * 128];
6.
7. for (int i = 0; i < 64; i++)
8.   subkeys2[i + threadIdx.x * 64] = key[i + 64 + tid * 128];
9.
10. for (size_t i = 1; i < 13; i= i+2) {
11.   pipo_addRoundKey(X, subkeys + threadIdx.x * 64);
12.   pipo_sBoxLayer(X);
13.   pipo_rLayer(X);
14.   pipo_update(X, i);
15.   pipo_addRoundKey(X, subkeys2 + threadIdx.x * 64);
16.   pipo_sBoxLayer(X);
17.   pipo_rLayer(X);
18.   pipo_update(X, i+1);
19. }
20. pipo_addRoundKey(X, subkeys + threadIdx.x * 64);
21. pipo_sBoxLayer(X);
22. pipo_rLayer(X);
23. pipo_update(X, 13);
24. pipo_addRoundKey(X, subkeys + threadIdx.x * 64);
```

Fig. 9. PIPO encryption of the proposed method

IV. 성능 평가

RTX 3060에서 Visual Studio 환경에서 구현하였으며 블록당 스레드 수와 그리드 당 블록 수를 조절하며 성능을 측정하였다. Table 1. 은 제안기법과 이전 연구 [7]과 비교 결과이다. 카운터 기반 키 탐색기법을 적용하지 않았을 때 Fig. 10. 과 같이 RTX 3060에서 그리드 당 블록 수 1024, 블록당 스레드 수 64에서 초당 약 3.6177억회 암호 연산을

Table 1. Comparison of implementation results.

Reference	GPU	Throughput
[7]	RTX 3070	159 million/s
bltslicing (our)	RTX 3060	361 million/s
bitslcing+ conter (our)	RTX 3060	19500 million/s

수행하였다. 블록당 스레드 수가 많을수록 그리드 당 블록 수가 많을수록 높은 성능을 보였다. 이전 연구인 [7]의 PIPO 암호화 병렬 구현에서 RTX 3070에서 초당 약 1.59억회 암호 연산을 처리한 결과보다 높은 처리량을 보였다. 구현 목적과 GPU의 성능

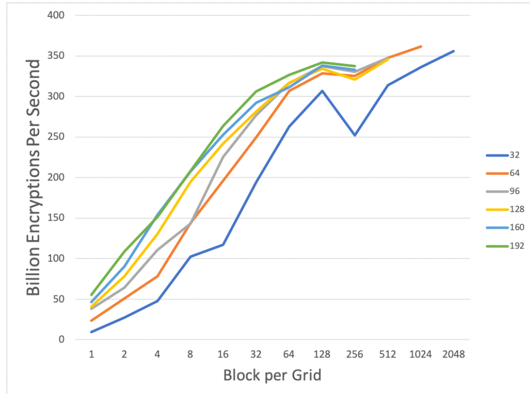

Fig. 10. Before applying the counter-based key search method, Comparison of the number of encryptions per second according to the number of blocks per grid and the number of threads per block. The x-axis is the number of blocks per grid, the y-axis is 1 billion encryptions per second, and the legend is the number of blocks per thread.

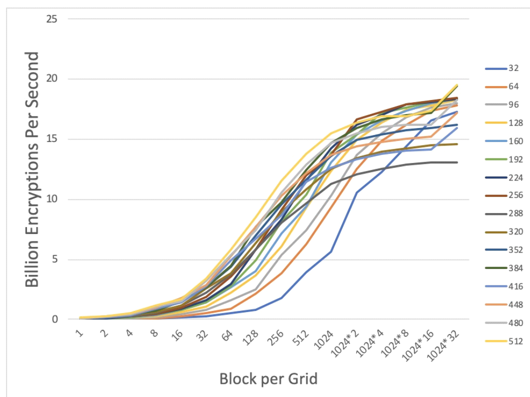

Fig. 11. After applying the counter-based key search method, Comparison of the number of encryptions per second according to the number of blocks per grid and the number of threads per block. The x-axis is the number of blocks per grid, the y-axis is 1 billion encryptions per second, and the legend is the number of blocks per thread.

차이는 있지만 비교적 큰 성능 향상을 보였다. 결과적으로 PIPO는 제안하는 고도의 비트 슬라이싱 기법을 적용하였을 때 이전보다 높은 처리량을 달성할 수 있었다. 같은 환경에서 카운터 기반 키 탐색기법을 적용하였을 때 성능은 Fig. 11. 과 같다. 그리드 당 블록 수 32,768, 블록 당 그리드 수 512에서 초당 약 195.685억 회의 암호연산을 수행하였다. 카운터 기반 키 탐색기법은 고도의 비트 슬라이싱 구현으로 발생하는 CPU에서 GPU로 평문을 전송할 때 하므로 많은 지연을 효과적으로 제거하였다.

V. 결 론

본 논문에서는 GPU 상에서 경량 블록 암호 PIPO를 최적 구현하였다. 비트 슬라이싱 기법을 적용한 최적화 구현과 GPU 요소를 최대한 사용하여 결과적으로 RTX 3060에서 그리드 당 블록 수 65536, 블록당 그리드 수 96에서 초당 약 403,463,100회의 높은 처리량을 달성하였다. 추후 연구에는 추후 연구에는 더 다양한 환경과 구현기법의 성능 비교와 다른 32비트 프로세서상에서 적용하여 효율적으로 동작하는지 확인하고자 한다.

References

[1] Lee, Wai-Kong, Bok-Min Goi, and Raphael C.W. Phan. "Terabit encryption in a second: Performance evaluation of block ciphers in GPU with Kepler, Maxwell, and Pascal architectures," Concurrency and Computation: Practice and Experience, 31(11), e5048, Jun. 2019.

[2] Hajihassani, Omid, et al. "Fast AES i mplementation: A high-throughput bit sliced approach," IEEE Transactions o n parallel and distributed systems, 30 (10), 2211-2222, Oct. 2019.

[3] Gupta, Naina, et al. "Pqc acceleration using gpus: Frodokem, newhope, and kyber," IEEE Transactions on Parallel and Distributed Systems, 32(3) 575-58 6, Mar, 2020.

[4] Lee, Wai-Kong, and Seong-Oun Hwan

g. "High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of Things applications," IEEE Transactions on Services Computing, Aug. 2021.

〔5〕 Kim, H. et al., "PIPO: A Lightweight Block Cipher with Efficient Higher-Order Masking Software Implementations," Information Security and Cryptology – ICISC 2020. ICISC 2020. Lecture Notes in Computer Science(), vol 12593. Springer, Cham, pp. 00-122, Feb. 2021.

〔6〕 In-yeung Kim, Byoung-jin Seok and Chang-hoon Lee "A Study of Fast Implementation of Korea Block Ciphers PIPO, HIGHT, and CHAM," Journal of Digital Contents Society. 22(12), pp 2063-2075, Dec, 2021.

〔7〕 S.W. Eum et al., "Optimized Implementation of Block Cipher PIPO in Parallel-Way on 64-bit ARM Processors," KIPS Transactions on Computer and Communication Systems, vol. 10, no. 8, pp. 223–230, Aug. 2021.

〔8〕 Kwak, Y., Kim, Y., Seo, S.C., "Parallel Implementation of PIPO Block Cipher on 32-bit RISC-V Processor," Information Security Applications, WISA 2021, Springer, vol 13009, pp. 183-193, Oct. 2021.

〔9〕 Jang K, Song G, Kwon H, Uhm S, Kim H, Lee W-K, Seo H., "Grover on PIPO," Electronics. 2021, 10(10):1194, May. 2021.

〔10〕 Lim S.-J., W.-W. Kim, Y.-J. Kang, and H.-J. Seo, "Implementation and performance evaluation of PIPO lightweight block ciphers on the web," Journal of the Korea Institute of Information and Communication Engineering, vol. 26, no. 5, pp. 731–742, May. 2022.

〔11〕 Kwon, H. et al., "Parallel implementation of the block cipher PIPO on GPU environment," Conference on Information Security and Cryptography-Summer 2021, Jun. 2021.

〔12〕 Biham, Eli., "A Fast New DES Implementation in Software," International Workshop on Fast Software Encryption, pp. 260-272, Jan. 1997.

〔13〕 Tezcan, Cihangir, "Key lengths revisited: GPU-based brute force cryptanalysis of DES, 3DES, and PRESENT," Journal of Systems Architecture, vol. 124, Mar. 2022.

〈저자소개〉

김 현 준 (Hyun-Jun Kim) 학생회원
2019년 3월: 한성대학교 IT응용시스템공학부 졸업
2021년 3월: 한성대학교 IT융합공학부 석사
1996년 3월~현재: 한성대학교 정보컴퓨터공학과 박사과정
〈관심분야〉 암호화구현, 부채널분석

엄 시 우 (Si-Woo Eum) 학생회원
2021년 3월: 한성대학교 IT응용시스템공학부 졸업
2021년 3월~현재: 한성대학교 IT융합공학부 석사과정
〈관심분야〉 암호구현, 정보보안

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