2021 PIPO 경진대회 결과제출

2021. 05. 31

참가분야 표시	**Track 1**	
	Track 2	
	Track 3	○
	학부 or 일반	일반

* 참고 : **참가 분야 학부는 참가자 전원 학부생에 해당함.**
* 팀 구성은 최대 3인으로 제한함.

지도교수	**성 명**	서화정
	소속	한성대학교 IT 융합공학부
	휴대폰	010-9350-3118
	E-mail	hwajeong84@gmail.com

참가자 1	**성 명**	김현지
	소속	한성대학교 IT 융합공학부
	휴대폰	010-2726-2859
	E-mail	khj1594012@gmail.com

참가자 2	**성 명**	박재훈
	소속	한성대학교 IT 융합공학부
	휴대폰	010-6216-1757
	E-mail	p9595jh@gmail.com

참가자 3	**성 명**	양유진
	소속	한성대학교 IT융합공학부
	휴대폰	010-7604-9916
	E-mail	yujin.yang34@gmail.com

보고서

1. 서론

국산 경량 블록암호인 PIPO 알고리즘은 병렬적으로 연산이 가능하여 효율적인 비트슬라이스 기법이 S-Layer에 자체적으로 구현되어 있습니다. 따라서 AVX2와 CUDA를 통해 PIPO 블록암호를 병렬 구현해보았으며, 기존 PIPO 구현물과의 비교를 통해 성능 향상을 확인하였습니다. 또한, 최근 빅데이터와 딥러닝 기술이 발전하면서 많은 데이터 셋을 저장해주는 대규모 데이터베이스의 필요성과 데이터베이스 접근 속도 및 메모리 사용량에 관한 부분들에 대한 중요성이 높아졌습니다. AVX2-PIPO16-FF1을 활용하여 대규모 데이터에 대한 병렬 연산과 형태 보존암호의 특징인 데이터 입출력 길이 및 형태 보존을 통해 효율적인 데이터베이스 암호화가 가능하도록 응용해보았습니다.

2. 관련연구

2.1 PIPO Blockcipher

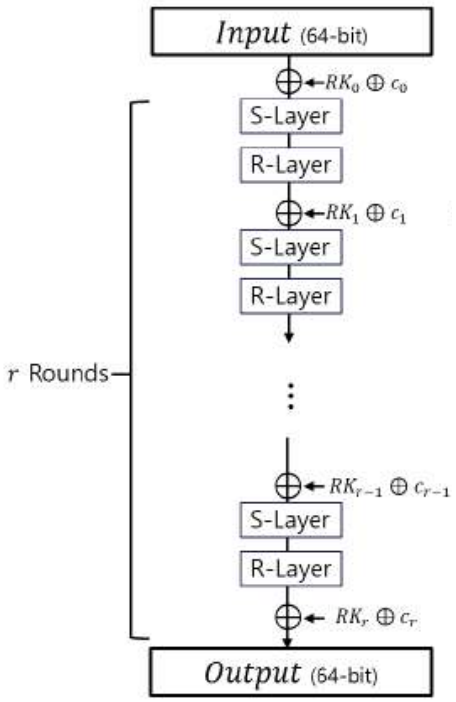


그림 1. PIPO 구조

PIPO는 ICISC'20에서 발표된 국산 경량 블록 암호입니다. 64-bit의 평문을 암호화 할 수 있으며, PIPO 64/128은 13 라운드이고, PIPO 64/256은 17번 라운드입니다. 그림 1에서 알 수 있듯 PIPO는 각 라운드마다 라운드키 RK_r와 c_r를 XOR 연산하는 Key Add와 S-box 치환을 수행하는 S-Layer, Rotation 연산해주는 R-Layer로 구성됩니다.

S-Layer의 경우 Bitslice와 Lookup 테이블 방식으로 구현되어 있습니다. 두 가지 방식 중 Bitslice는

비트를 병렬적으로 연산할 수 있어서 효율적인 구조입니다. S8은 11개의 비선형 비트 연산과 23개의 선형 비트 연산만 포함하기 때문에 효율적인 비트 슬라이스 구현을 제공합니다.

2.2 병렬구현

병렬 연산은 프로세스, 쓰레드 등의 단위를 이용하여 여러 연산을 한 번에 수행하는 것을 말합니다. 전통적으로 데이터 처리에 있어서 CPU를 이용할 때, 하나의 연산이 끝나야 다음으로 넘어가는 순차적 연산이 이용되었으며, 이것은 처리량이 많은 작업에서는 의미가 컸지만 다량의 단순 반복적인 연산에서는 상당히 비효율적이었습니다. 이에 따라 병렬 연산을 이용하여 이러한 작업을 처리하려는 시도가 지속적으로 생겼으며, 특히나 최근에는 딥러닝과 같이 다량의 연산을 하는 작업이 보다 많이 필요하게 되어 최적화에서의 병렬 연산의 중요도가 더욱 올라가고 있는 추세입니다. 본 구현에서는 AVX2 및 GPU를 이용하여 병렬 연산을 진행하였습니다.

2.3 형태보존암호

형태보존암호는 평문과 암호문의 크기를 동일하게 하는 것입니다. 기존 블록 암호의 경우 블록 하나의 크기를 64-bit라고 가정할 시, 입력값의 크기가 20-bit라고 하여도 암호화를 거치고 나면 64비트로 바뀌게 됩니다. 빈 공간은 패딩을 하여 채우게 되며, 이렇게 패딩된 공간은 공간의 낭비일 뿐만 아니라, 패딩인지 확인하는 작업 또한 필요하게 되어 다량의 데이터를 암호화 하여 보관할 때에는 그만큼 비용적인 소모가 큰 부분입니다. 이를 해결하기 위해 형태보존암호가 등장하게 되었습니다. FF1, FF3, FPA 등 다양한 종류가 있으며, 형태보존암호는 주로 비용적인 측면 때문에 쓰이고 있습니다. 대표적으로 카드회사인 비자(VISA)의 경우 자체적으로 개발한 형태보존암호인 VFPE를 사용하고 있습니다.

형태보존암호는 라운드 키 이외에도 트윅 값을 추가적으로 이용하여 암호화를 진행합니다. 트윅 값은 평문에서 암호화에 필요하지 않은 부분을 주로 이용합니다. 가령 주민등록번호를 암호화 한다고 하면, 앞 6자리는 암호화 하지 않아도 되기에 이것이 트윅 값으로 사용되는 방식입니다. 형태보존암호의 구현에는 주로 각 라운드의 결과가 다음 라운드에 영향을 미치는 방식으로 구성되어 보안성을 강화하는 파이스텔 구조가 활용됩니다.

2.4 데이터베이스 암호화

데이터베이스 암호화는 데이터베이스 접근 권한이 있는 서버와 사용자만이 복호화할 수 있게 데이터를 암호화하고 저장함으로써 정보 유출을 막는 데이터 보호 기술입니다. 데이터베이스 암호화 방식으로는 크게 칼럼(column) 암호화 방식과 블록(Block) 암호화 방식이 있습니다.

그림 2. 데이터베이스 암호화 방식 (암/복호화 위치에 따른 분류)

그림 2를 보면 칼럼 암호화 방식과 달리 블록 암호화 방식이 주로 데이터 저장소에서만 암호화 상태인 것을 확인할 수 있습니다. 이는 마스킹과 같은 암호화 외의 다른 조치 없이 블록 암호화 방식을 사용하고 있을 때 공격자가 애플리케이션서버나 DB서버를 공격할 경우, 칼럼 암호화 방식보다 데이터 유출의 위험이 더 크다고 해석 할 수 있습니다.

이러한 칼럼 암호화 방식은 암·복호화 되는 위치에 따라 API, 플러그인, 하이브리드 방식으로 나눌 수 있습니다. 항목에 따른 세 가지 암호화 방식 비교는 아래 표 1에서 볼 수 있습니다.

표 1. 칼럼 암호화 방식 비교

항목	API	플러그인	하이브리드
암·복호화	애플리케이션 서버	DB 서버	애플리케이션 서버, DB 서버
DB 서버 부하 발생	없음	높음	보통
구축비용	낮음	보통	높음
암호 알고리즘	SHA-256/384/512, 키 길이 128 비트 이상의 AES, TDES, SEED, ARIA 등 지원		

세 가지 방법 중 데이터를 삽입할 때 데이터 자체를 암호화하여 삽입하는 'API 방식'의 경우 애플리케이션 서버에서 암·복호화를 수행하기 때문에 DBMS 상에서의 부하를 분산시켜주고, 애플리케이션 서버에서 DB계정이 탈취되더라도 데이터 자체가 암호화 되어 있기에 데이터 유출될 위험이 가장 적습니다. 또한, API 방식은 구축비용이 세 방법 중 가장 저렴하다는 점에서 장점을 갖습니다.

3. 제안 기법

국산 경량 블록암호 알고리즘인 PIPO를 활용하여 다음과 같이 3가지 응용 기법을 제안합니다.

3.1 AVX2를 활용한 PIPO 병렬 구현

AVX2는 앞서 언급하였듯이 256-bit의 레지스터를 사용하여 256-bit 데이터를 하나의 명령어로 처리 가능한 SIMD 명령어 집합 중 하나입니다. AVX2를 통한 병렬 구현을 위해 룩업테이블 방식인 TLU 구현이 아닌 bitslice 방식을 사용하였습니다. 또한, 효율적인 병렬 구현을 위한 명령어 사용, 정렬, 로테이션을 적용하였습니다. PIPO는 다음과 같이 Key schedule, Key add, S-Layer, R-Layer로 구성되어 있습니다. 세부 기법 설명에 앞서 구현에 사용된 AVX2 명령어는 표 2와 같습니다.

표 2. AVX2 instruction set for AVX2-PIPO

Instruction	Description
_mm256_loadu_si256	Load 256-bit of integer data from memory
_mm256_set1_epi16	Broadcast 16-bit integer to all elements of dst
_mm256_xor_si256	bitwise XOR of 256 bits
_mm256_and_si256	bitwise AND of 256 bits
_mm256_or_si256	bitwise OR of 256 bits
_mm256_andnot_si256(a,b)	bitwise NOT of 256 bits in a and then AND with b
_mm256_slli_epi16/ _mm256_srli_epi16	Shift packed 16-bit integers in a left/right

3.1.1 Plaintext Arrangement & Key schedule

PIPO는 암호화 과정에서 8-bit S-box 연산을 수행합니다. 즉, 총 8byte길이의 평문에서 한 byte씩을 동일한 레지스터에 로드하게 되므로 평문 32개를 동시에 암호화 할 수 있게 됩니다. 그러나 효율적인 병렬 로테이션 연산을 위해 본 구현에서는 8개의 256-bit 레지스터를 활용하여 64-bit 평문 16개를 입력받습니다. 따라서 AVX2-PIPO에서는 바이트 단위로 병렬 연산을 수행하기 위해 라운드 시작 전에 그림 3,4 와 같이 바이트 단위로 기존 평문들의 형태를 변환해주어야 합니다.

그림 4에서 볼 수 있듯이, 각 평문은 8바이트로 구성되며, 이는 바이트 단위로 나뉘어 temp 배열에 입력됩니다. 이후, _mm256_loadu_si256 명령어를 통해 temp 배열의 각 행을 하나의 (동일한) 256-bit 레지스터에 로드합니다. 즉 16개의 평문에 대한 각 바이트들은 각각의 256-bit 레지스터에 16-bit 단위로 저장되고, 해당 레지스터들은 0라운드의 Key Addition에 입력으로 사용됩니다.

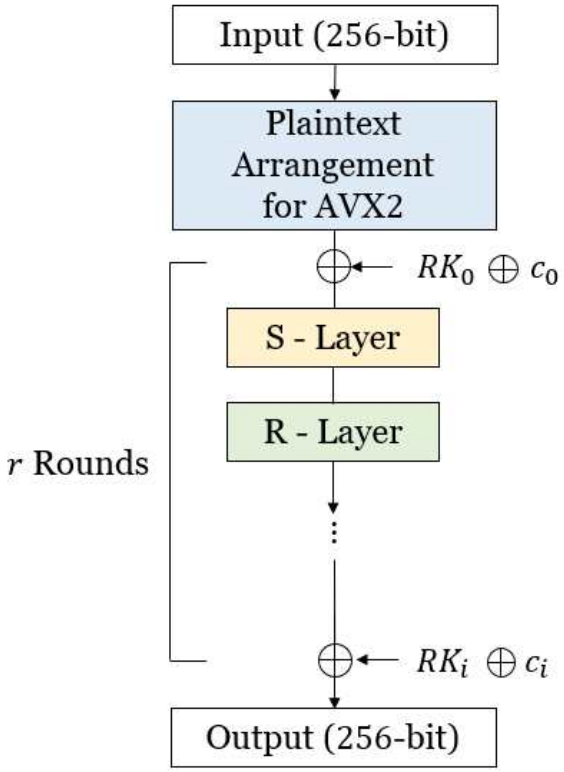


그림 3. AVX2-PIPO

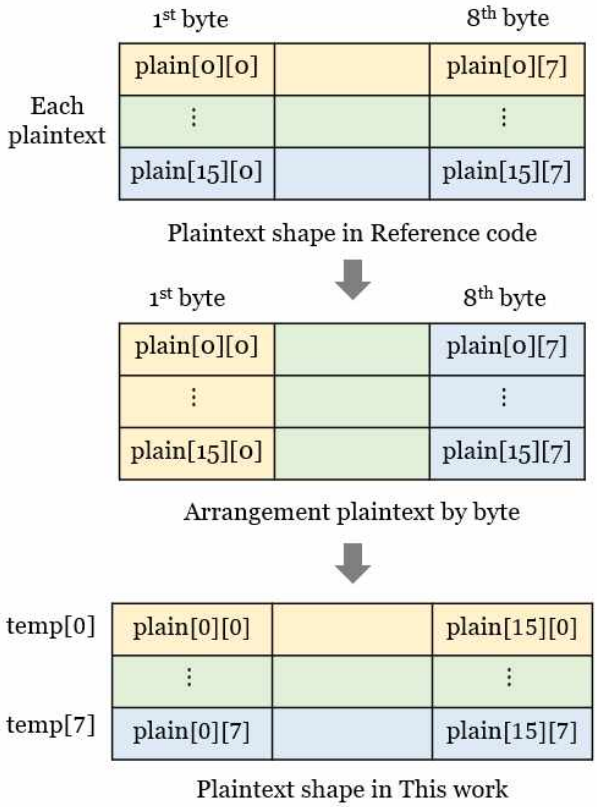


그림 4. AVX2를 활용한 병렬 구현을 위한 Plaintext Arrangement

3.1.2 Key Schedule

PIPO의 키 스케줄은 모든 평문에 동일한 키가 사용되는 형태이므로 병렬 처리를 하지 않고 기존 PIPO와 동일하게 생성합니다. 그러나 KeyAdd 과정에서 256-bit 레지스터에 저장된 평문과 비트단위 XOR을 수행해야하므로 라운드키 또한 256-bit 레지스터에 로드해야합니다. 즉, 평문의 n번째 바이트와 라운드 키의 n번째 바이트가 연산되어야 하고, 여러 평문에 대해 동일한 라운드 키가 사용되어야 하므로 동일한 8-bit 단위의 데이터를 256-bit 레지스터에 저장하는 함수를 사용하였습니다.

3.1.3 Key Add

키 덧셈 과정은 앞서 언급한 두 256-bit 레지스터들 (입력 평문 및 라운드 키)을 XOR하는 과정입니다. 즉, 256-bit 레지스터와 bitwise XOR 되므로 평문의 각 바이트에 대응하는 라운드 키 바이트들을 하나의 레지스터에 채워야합니다. 이 때, 라운드 키의 한 바이트를 전체 16 바이트에 set1 명령어 (표 2 참조)를 통해 세팅하며, 그 결과 8번의 연산으로 총 16개의 평문에 대한 키 덧셈이 가능해집니다. 그림 5에서의 T256은 연산의 중간 값들을 저장하는 256-bit 레지스터 (0라운드 이전에는 평문이 로드된 상태)이고, RK256은 256-bit 레지스터에 저장된 라운드 키입니다.

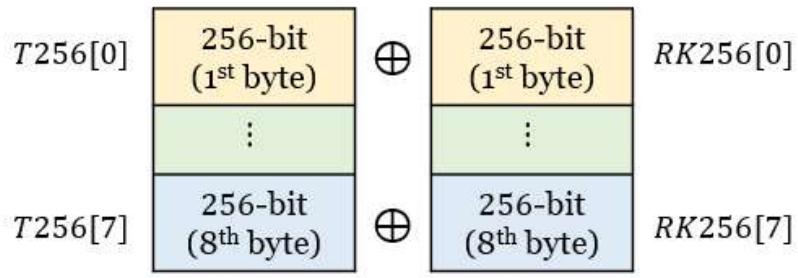


그림 5. AVX2-PIPO16의 Key Add 연산

3.1.4 S-Layer

PIPO의 S-Layer의 경우 lookup table인 TLU와 bitslice 기법으로 구현이 가능하지만, 효율적인 병렬 연산을 위해 제안 기법에서는 bitslice 방식을 사용하였습니다. Bitslice S-Layer의 경우 각 바이트들 간의 XOR, AND 연산 등을 통해 S-box가 구현된 것입니다. 본 구현에 사용되는 256-bit 레지스터의 경우 각 평문의 각 바이트들을 모아놓은 것이므로 기존 PIPO와 동일한 인덱스를 사용하여 접근할 수 있으며, 이를 통해 16개의 평문에 대한 병렬 연산이 가능합니다. 그러나 AVX2에는 S-Layer에 사용되는 NOT 명령어가 존재하지 않습니다. 따라서, NOT 연산을 구현하려면 최소 두 개의 명령어를 합쳐야 하므로, AVX2가 지원하는 _mm_andnot_si256(a,b) 명령어 (표2 참조)를 사용하였습니다. 해당 명령어는 하나의 레지스터(a)의 값들을 반전한 후 256-bit의 mask (b)를 AND 합니다. NOT 연산 값을 보존하기 위해, NOT 연산을 수행할 바이트들이 담긴 레지스터(T256[2])를 입력하고, 256-bit mask 전체를 1로 설정하였습니다

3.1.5 R-Layer

R-Layer는 rotation 연산을 수행하는 구간입니다. 앞서 말씀드렸듯이, PIPO는 바이트 단위 연산을 수행하므로 동일한 바이트에 대해 동일한 연산이 수행됩니다. 따라서 256-bit 레지스터를 최대로 활용하기 위해서는 32개의 평문을 동시에 연산해야합니다. 그러나 AVX2가 8-bit Rotation 연산을 위한 8-bit Shift 연산을 지원하지 않기 때문에 R-layer의 전후에서 8-bit 단위로 나눠진 256-bit 레지스터를 16-bit 단위로 바꾸어주어야 합니다. 이 과정에서 추가적인 레지스터가 요구되며, 해당 레지스터들에 16-bit로 바뀐 값들을 다시 로드해주어야 하므로 실제 8-bit 로테이션 연산에는 불필요한 부분들이 추가됩니다. (해당 방법도 구현하였습니다.)

따라서 본 구현물에서는 이를 방지하기 위해 평문 블록 16개에 대해 병렬처리하여 불필요한 과정을 제거하였습니다. 사용된 로테이션 연산은 그림 6과 같이 두 번의 Shift 연산과 1번의 OR 연산으로 구성됩니다. 이 과정에서 원본 값을 보존하기 위한 레지스터가 1개 추가적으로 필요합니다. 로테이션 연산을 위한 shift를 위해 ANDMASK를 설정합니다. 예를 들어 왼쪽으로 7비트 rotation을 수행할 경우, 7비트 왼쪽 shift와 1비트 오른쪽 shift를 수행한 후, 두 결과를 OR 연산하여야 합니다. 이 과정에서 shift 연산을 하면서 사라지게 될 부분은 ANDMASK에서 0으로 지정하여 해당 마스크를 대상 비트에 XOR 합니다. 두 shift 결과 값을 OR하면 로테이션 연산에 대한 결과 값이 산출됩니다. 또한, 모든 라운드가 종료된 후,

최종적으로는 전체 256-bit 레지스터 (저장된 값들은 16-bit 단위로 나뉘어져 저장되어 있음)에서 각 16-bit 중 하위 8-bit만을 사용합니다.

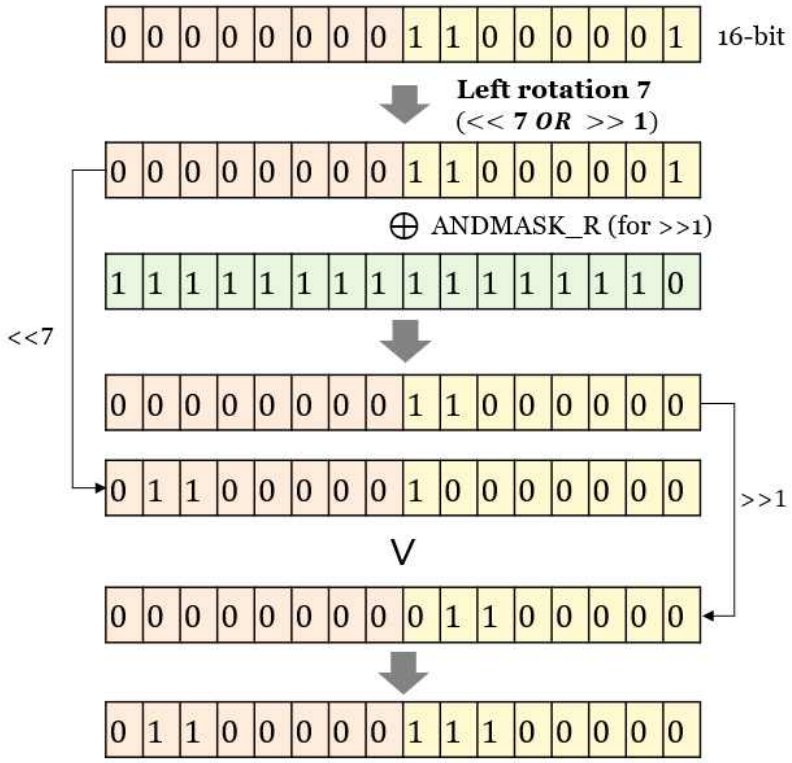


그림 6. AVX2-PIPO16를 위한 효율적인 16-bit Rotation

3.2 FF1 with AVX2-PIPO16를 통한 데이터 베이스 암호화

3.2.1 FF1

형태보존암호 중 하나인 FF1은 라운드 함수 내부에서 AES-128 암호화를 사용하는데, AES-128을 대신하여 AVX2-PIPO16을 적용하였습니다. 라운드 함수 내부에서 AES-128이 사용되므로 AES 암호화를 위한 128-bit 입력이 필요하며, AVX2-PIPO16는 16개의 64-bit 입력이 필요합니다. 즉, AVX2-PIPO16을 라운드 함수에 적용하기 위해서는 기존 FF1에서의 입력인 32byte 입력이 8개가 필요합니다. 또한, PIPO의 마스터키로부터 PIPO key schedule 과정을 통해 라운드키가 생성되고, 이는 FF1의 라운드 함수의 키 값으로 사용됩니다.

그림 7은 전체적인 구성도입니다. 라운드 함수에서 AES-128이 사용되는 부분은 PRF와 CIPH 함수입니다. PRF의 경우 입력 길이가 16byte로 고정된 P를 AES-CBC mode의 초기 블록으로 사용합니다. 따라서 radix, tweak 등의 값을 활용하여 설정된 P를 AES-CBC에 평문으로 입력하여 암호문을 얻어내고, 해당 암호문을 R에 저장합니다. PRF의 경우 하나의 초기 블록을 입력으로 사용하기 때문에 병렬 구현된 AVX2-PIPO16 대신 기존의 AES-128을 사용하였습니다. CIPH의 경우는 PRF를 통해 초기화된 R과 tweak과 b값(잘린 평문의 한쪽의 길이) 등을 통해 초기화 된 Q를 XOR합니다. 이 과정에서 AES-128에 입력하기 위해 Q를 16 byte씩 접근하여 연산하게 됩니다. 이로 인해 여러 개의 평문 블록이 생기게 되

며, 해당 128-bit 블록들을 8개씩 수집하여 AVX2-PIPO16의 입력으로 사용합니다. 즉, 8개의 평문 블록이 AVX2-PIPO16에 입력되고 16개의 64-bit 블록으로 변환되어 병렬적으로 처리됩니다. 이후에는 기존의 FF1의 프로세스를 수행합니다.

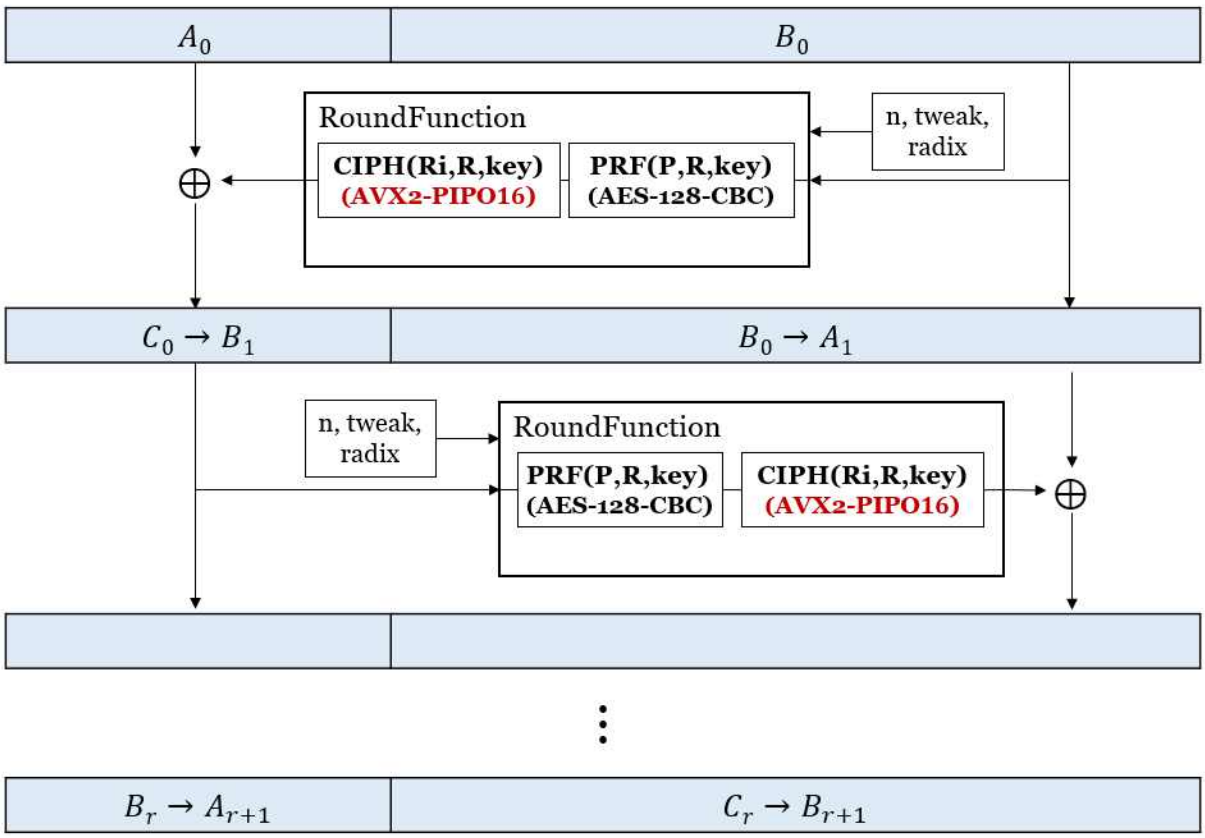


그림 7 AVX2-PIPO를 적용한 FF1 구성도

3.2.2 AVX2-PIPO16-FF1을 활용한 데이터베이스 암호화

MySQL의 경우 기본적으로 AES(CBC, ECB모드) DES와 같은 양방향 암호화와 MD5, SHA1, SHA2와 같은 단방향 암호화 알고리즘을 제공합니다. SHA2('평문', Hash 길이), AES_ENCRYPT('평문', 'key', 'IV') / AES_DECRYPT('암호문', 'key', 'IV')와 같이 MySQL에서 제공하는 암호화 함수를 활용하면 데이터를 간단하게 암·복호화할 수 있습니다. 다음과 같은 방법으로 암호화하는 방식은 데이터를 암호화한 후 DB서버로 넘겨주는 방식이기 때문에 애플리케이션 서버에서 암호화하는 'API 방식'이라 볼 수 있습니다.

MySQL은 다양한 언어로 구현이 가능하다는 장점을 가지고 있기에 C언어로도 구현이 가능합니다. C언어상에서 mysql.h 헤더파일을 추가하고 mysql_query()라는 함수에 SQL문법에 맞는 Query를 입력하면 테이블 생성과, 데이터 추가/삭제와 같은 데이터베이스 관리 작업이 가능합니다.

데이터를 암호화하여 추가할 때, 제공된 암호화 함수를 사용하는 경우 평문의 길이와 상관없이 무조건 128/256비트의 고정된 형태로 암호화되기 때문에 데이터베이스 낭비를 초래합니다. 이 문제를 보완하기 위하여 형태보존암호인 AVX2-PIPO를 사용한 함수를 만들어 데이터베이스에 데이터를 형태 그대로 암호화하여 삽입할 수 있도록 하였습니다. API 방식을 사용하기 때문에 AVX2-PIPO를 활용한

암·복호화는 애플리케이션 서버에서 수행합니다. 전체 과정은 아래 그림 8에서 확인할 수 있습니다.

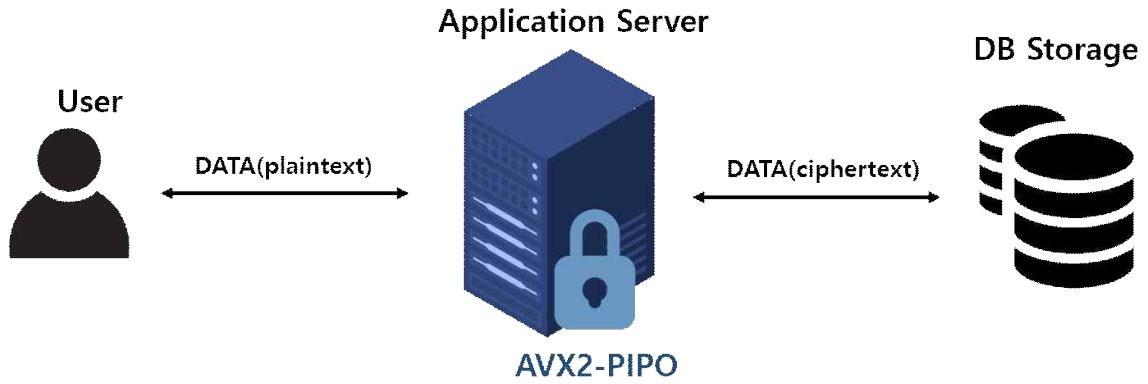


그림 8 API Database Enc/Decryption Process

3.3 GPU에서의 PIPO 병렬 구현

GPU를 이용하여 병렬 연산을 구현하기 위해, NVIDIA CUDA를 이용하였습니다. 사용한 GPU는 GTX 1060 3GB이며, CUDA 버전은 11.3입니다.

PIPO의 S-Layer는 Bitslice, TLU 두 가지로 구현되어 있는데, 두 버전 모두 GPU 병렬 구현을 하였으며 본 기법에서는 TLU에 대해 자세히 다룹니다. GPU 구현을 위해 기존의 연산 함수들을 디바이스 코드로 바꾸었으며, 연산이 한 번 일어나는 사이즈가 2라고 할 때 이 연산의 횟수를 다양하게 늘려 테스트에 이용하였습니다. 연산의 과정은 먼저 평문과 암호문을 CPU에서 생성한 뒤, cudaMalloc을 통해 GPU에서 이용 가능하도록 GPU 변수를 새로 만들었습니다. 그리고 call_work 함수에서 마스터키와 라운드 키를 ROUND_KEY_GEN_gpu 글로벌 함수를 통해 정의하였으며, GPU 연산에 필요한 SBOX 및 SBOX_INV도 생성하였습니다. SBOX와 SBOX_INV는 GPU 디바이스에서의 사용을 위해 cudaMalloc을 통해 Sbox_gpu 변수를 생성하였습니다. 기존의 래퍼런스 코드에서는 단순 테스트를 위해 필요한 평문, 암호문, 마스터키, Sbox, Sbox_inv가 모두 전역변수로 선언이 되어 있었으나, 이 변수들은 전부 CPU에서 선언된 것이기에 GPU 디바이스에서의 사용을 위해서는 GPU에서의 메모리 스왑이 필요합니다. 따라서 상기하였듯 고정 값인 Sbox와 Sbox_inv는 cudaMalloc을 통해 할당한 뒤 각 연산들에서 이용하였습니다. 크기 또한 정해져 있으므로 단순히 배열 첫 값(Sbox 및 Sbox_inv의 포인터 변수)만 있으면 모든 내부 원소에 접근할 수 있기에 한 번 할당해놓으면 다른 디바이스 함수들에 파라미터로 추가해놓으면 되어, 사전에 값을 상수로 할당해놓은 뒤 다른 디바이스 함수들에서 이용하였습니다.

다른 값들의 경우 기본적으로 정의된 길이 2의 값을 바탕으로 이용하였습니다. 모든 연산은 병렬로 수행되기 때문에, 길이 2의 값들이 각각 기존 방식과 동일하게 암호화가 진행됩니다. 그래서 라운드 키는 모든 각각의 연산들에 대해 동일한 값을 이용합니다. ROUND_KEY_GEN_gpu 함수에서 마스터키를 바탕으로 라운드 키를 생성하게 되며, Sbox 및 Sbox_inv와 동일하게 라운드 키도 하나의 고정된 값을 생성한 뒤 연산에 읽기 작업만을 통해 이용하게 됩니다. 라운드 키의 생성은 CPU가 아닌 GPU 디바이스 내에서 실행됩니다. 파라미터로 입력받은 마스터키를 바탕으로 생성되며, 마스터키는 CPU에서 생성한 뒤 GPU 디바이스에 값을 넘겨주는 방식입니다.

평문은 응용을 위해 입력받은 값을 바탕으로 설정된다. 입력받은 값을 기본적으로 정의된 길이인 2로

나눠 각 블록에서 연산을 수행하게 됩니다. 그렇기에 GPU에서 연산을 수행하는 과정은 각 블록에 대해서는 CPU와 동일합니다. 다만 훨씬 큰 사이즈의 연산에 있어 연산 속도에 있어 훨씬 큰 이점을 얻을 수 있습니다. 이렇게 생성된 변수들을 통해 글로벌 함수 work_gpu에서 GPU 연산을 수행하였으며, 이 함수를 호출할 때에는 연산이 수행되는 횟수 len을 SIZE (기본 사이즈 2)로 나눈 값을 블록 개수로, 그리고 각 블록 당 쓰레드의 개수는 1개로 설정하여 연산을 수행하였습니다. 따라서 연산은 총 len / SIZE번 수행되게 됩니다.

그림 9. Sample input data

만약 그림 9와 같은 데이터가 있다고 가정하면, CPU에서의 연산은 이 데이터를 두 자씩 잘라 총 4번의 연산을 수행하게 됩니다. 하지만 병렬 연산을 이용하면, 4번의 연산을 한 번에 실행시켜 4배의 이득을 얻게 됩니다. SIZE 단위씩 연산을 한다고 하였을 때 n개의 데이터가 있고 한 번의 연산에 m의 시간이 걸린다고 가정한다면, CPU에서는 (n / SIZE) * m의 시간이 걸리게 되고, GPU에서는 m의 시간만이 걸리는 것입니다.

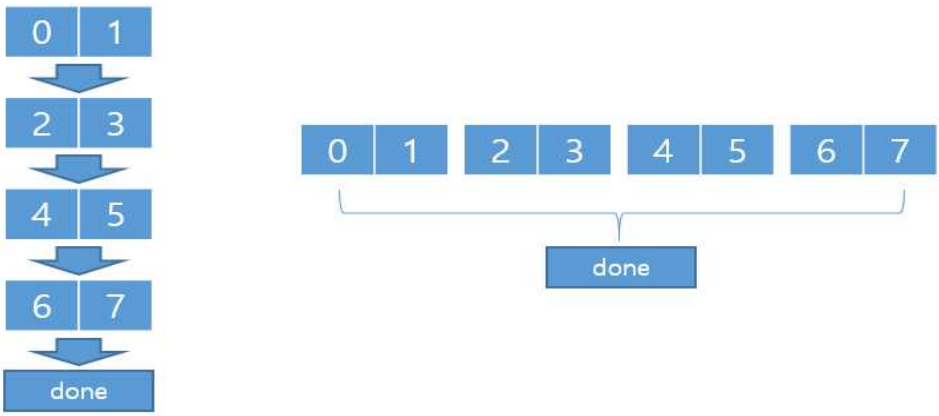


그림 10. Compare processing data between serially and parallelly

이에 대한 연산 과정을 정리하면 그림 10과 같습니다. 다만, 연산 속도의 경우 실제로는 GPU에 메모리를 할당해주고 연산이 끝났을 경우에는 결과를 CPU로 가져오는 과정이 필요하여 단순 비교보다는 더 오래 걸리게 됩니다.

4. 구현 결과 및 성능 평가

4.1 기존 PIPO와 AVX2-PIPO 비교

성능 평가를 위해 16개, 32개의 평문 블록을 병렬 처리한 구현물 (AVX2-PIPO16, AVX2-PIPO32)과 레퍼런스 코드에 대해 성능을 측정하였습니다. 또한, 본 실험은 MacBook Pro 2.6GHz 6core Intel Core i7 (16GB RAM) 상에서 수행되었으며, 컴파일 옵션 -O1과 -mavx2를 적용하였습니다. 성능 비교 방법으로는 연산 수행 시간 및 Cycle Per Byte (cpb)를 사용하였습니다.

4.1.1 구현 결과

3.1절의 제안 기법을 통해 AVX2를 적용한 PIPO를 구현하였으며, 레퍼런스 코드의 테스트 벡터를 사용하여 적절하게 구현되었는지 확인하였습니다. 그림 11과 같이 라운드 키는 모든 블록에 대해 동일하게 생성되었습니다. 또한, 평문에 모두 동일한 테스트 벡터를 적용하였을 때 그림 12와 같이 기존 PIPO의 마지막 라운드의 출력과 동일하게 생성되는 것을 볼 수 있습니다.

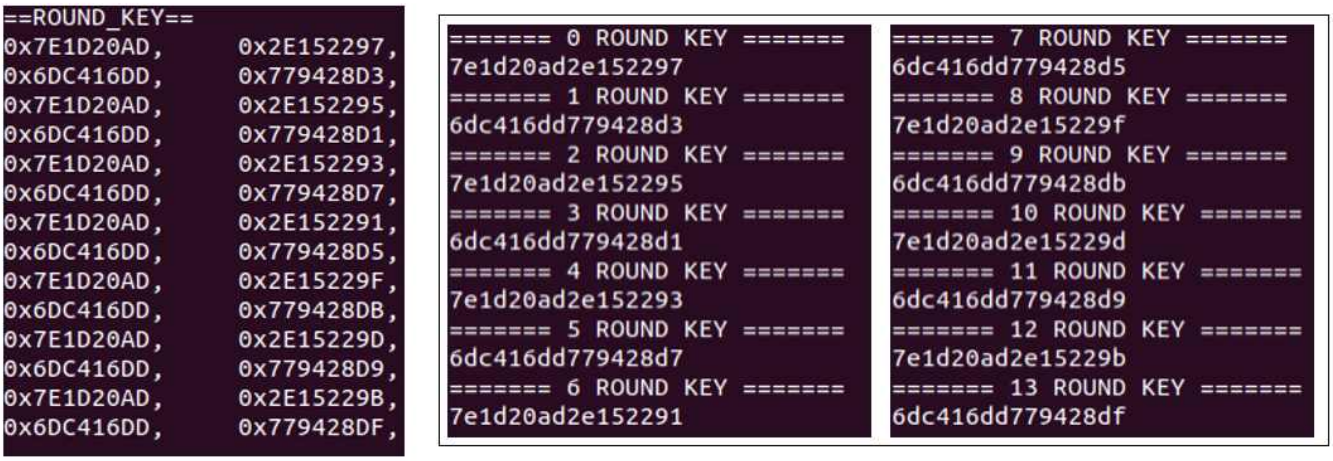


그림 11. 기존 PIPO와 AVX2-PIPO의 라운드 키 비교

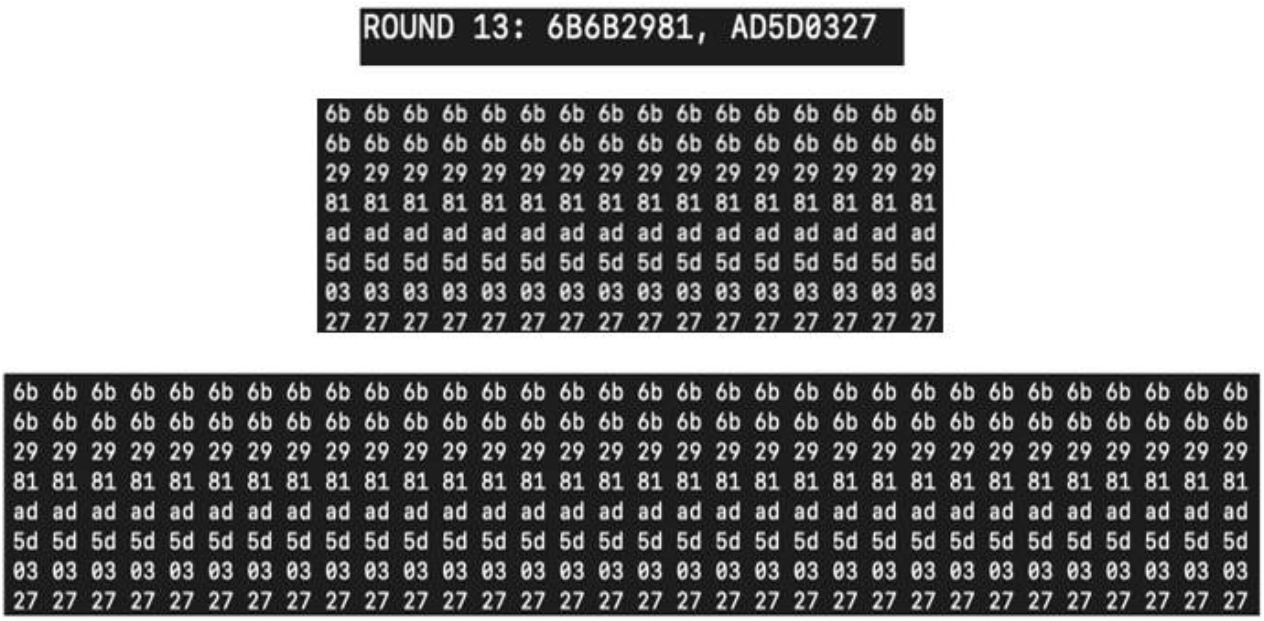


그림 12. 기존 PIPO(위)와 AVX2-PIPO(PT 16, PT 32)의 마지막 라운드 출력값 비교

4.1.2 속도 비교

표 3은 각 평문블록의 수에 따른 연산 수행 시간을 비교한 것입니다. Reference C code의 경우, 평문 블록 64개 이상부터 AVX2 구현물보다 속도가 저하되기 시작하였습니다. 64개 보다 적은 평문 블록 수를

처리할 경우, 병렬 연산을 위한 사전 작업 등으로 인해 병렬 구현물이 오히려 속도가 느린 것을 확인할 수 있었습니다. 또한, 평문 개수가 늘어남에 따라 병렬 구현물의 성능이 좋아졌으며, 16개의 평문 블록을 처리하는 경우 (AVX2-PIPO16)는 16만개 이상의 평문을 동시에 처리할 경우, Ref. C와 비교하였을 때 1.025배에서 5.943배로 성능이 크게 향상되었습니다. 이에 비해 더 많은 평문을 처리하지만 R-Layer에서 불필요한 전처리 과정이 필요한 AVX2-PIPO32 구현의 경우, 64개의 평문에 대해 1.003배, 3200만 개의 평문을 처리할 경우에 대해 1.801배의 성능 향상을 보였습니다. 따라서 AVX2-PIPO16 구현물의 경우가 가장 성능이 좋았으며 3200만개의 평문 블록에 대해 Ref.C에 비해 7.345배의 연산 속도를 보였습니다.

표 4는 본 구현물인 AVX2-PIPO32, AVX2-PIPO16과 레퍼런스 구현물의 cpb를 비교한 표입니다. 해당 결과 역시 AVX2-PIPO16의 성능이 가장 좋으며, Ref. C에 비해 약 7.34배의 성능향상을 보였습니다. AVX2-PIPO32의 경우는 Ref. C에 비해 1.8배의 성능향상이 있었으며, 두 구현물을 비교한 결과, AVX2-PIPO16 구현이 AVX2-PIPO32에 비해 4.07배 더 좋은 성능을 달성하였습니다.

표 3. Comparison Result Table (Speed) : Reference Code, AVX2-PIPO32 and AVX2-PIPO16 (-O1, unit : ms)

The number of PT	Ref. C	AVX2-PIPO32	AVX2-PIPO16
32	0.501	0.505	0.502
64	0.511	0.509	0.503
320000	54.647	32.876	9.003
3200000	451.136	260.624	73.538
32000000	4390.659	2437.528	597.735

표 4. Comparison Result Table (cpb): Reference Code, AVX2-PIPO32 and AVX2-PIPO16 (-O1, unit : cpb)

Ref. C	AVX2-PIPO32	AVX2-PIPO16
44.592	24.75	6.07

4.2 AVX2-PIPO16을 적용한 FF1

4.2.1 AVX2-PIPO16-FF1 구현 결과

그림 13은 AVX2-PIPO16를 FF1의 라운드 함수에 적용하여 암/복호화한 결과입니다.

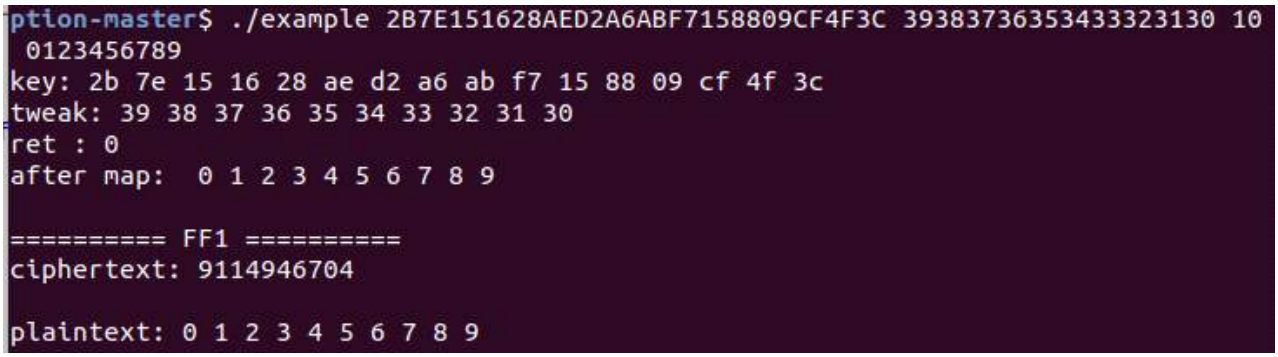


그림 13-(a). 평문 길이 10, 암호문 길이 10, radix 10

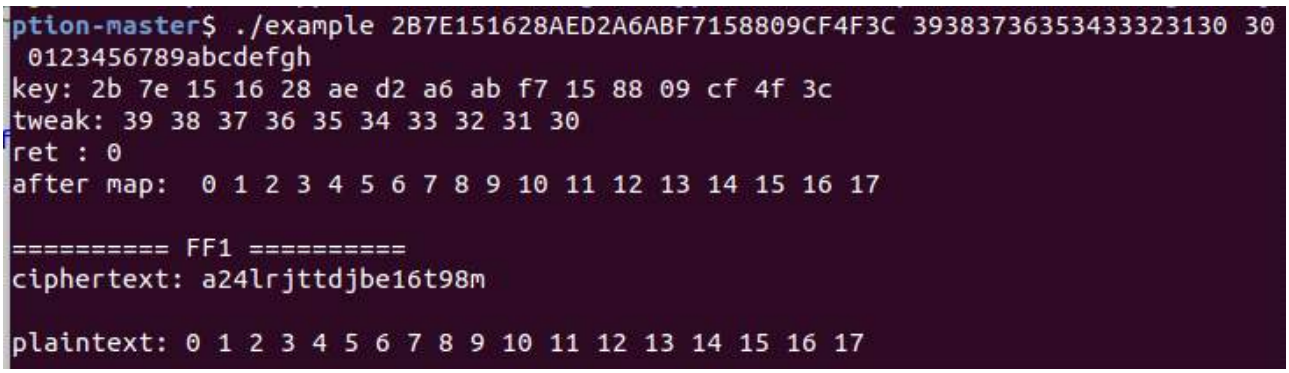


그림 13-(b). 평문 길이 18, 암호문 길이 18, radix 30

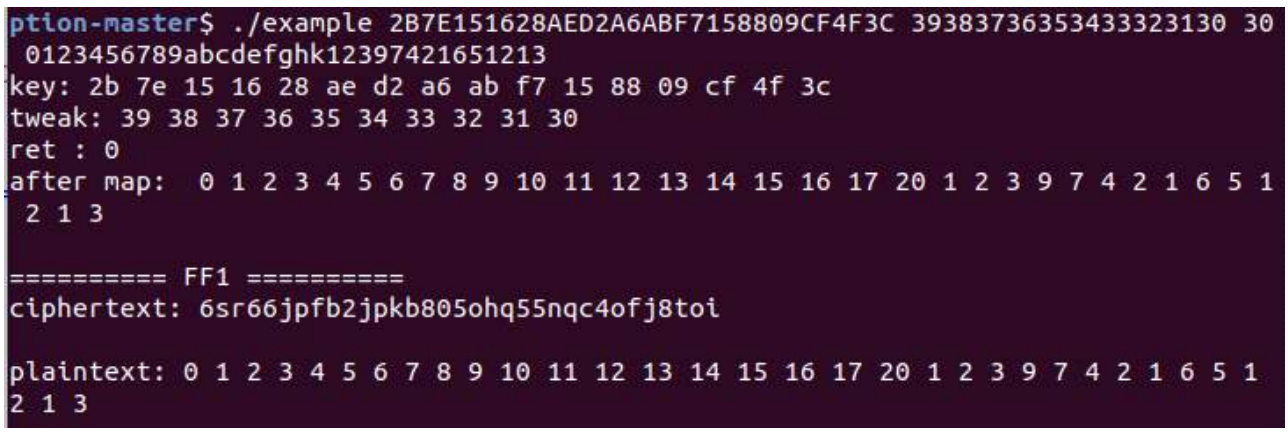


그림 13-(c). 평문 길이 33, 암호문 길이 33, radix 30

그림 13. AVX2-PIPO를 적용한 FF1을 통해 데이터 암호화 및 복호화

4.2.2 데이터베이스 암호화

가. 데이터베이스 암호화 결과

그림 14는 MySQL을 사용하여 API 방식으로 데이터를 암호화한 후 데이터베이스에 저장한 결과이며, DB내의 FF1_PIPO라는 테이블에 앞서 암호화한 암호문들 (그림 13-(a), (b), (c))이 저장되어 있는 것을 확인할 수 있습니다. 형태보존암호를 사용하였으므로 데이터베이스에 저장된 암호화 될 데이터의 크기가 암호 알고리즘의 평문 및 암호문 크기에 맞도록 패딩되지 않습니다. 따라서 동일한 길이의 입출력이 보장

되어 데이터베이스의 메모리 저장 공간의 낭비를 막을 수 있습니다. 또한, 원본 데이터와 동일한 형태를 가지기 때문에 해당 데이터가 저장될 데이터베이스의 스키마 변경이 필요하지 않습니다. 예를 들어, 사용자 아이디와 비밀번호를 저장하는 테이블에서 'A1234' 비밀번호를 저장하려고 할 때, 비밀번호 컬럼의 도메인이 알파벳과 숫자가 조합된 문자열일 경우, 해당 범위 내에서 암호화되어 길이 및 형태를 보존할 수 있기 때문에 추가적인 데이터베이스 스키마 변경이 필요하지 않은 것입니다. 따라서 데이터베이스 관리 비용 증가 및 시스템 수정이 거의 발생하지 않습니다.

```
mysql> SELECT * FROM FF1_PIPO;
+------+--------------------------------------+
| ID   | PW                                   |
+------+--------------------------------------+
| id05 | 9114946704                           |
| id05 | a24lrjttdjbe16t98m                   |
| id05 | 6sr66jpfb2jpkb805ohq55nqc4ofj8toi    |
+------+--------------------------------------+
3 rows in set (0.01 sec)
```

그림 14. AVX2-PIPO16-FF1으로 암호화된 데이터가 저장된 데이터베이스

나. MySQL 지원 암호화와 AVX2-PIPO16-FF1의 데이터베이스 암호화 비교

MySQL에서는 AES, SHA2, DES 등과 같은 암호화 체계를 쉽게 사용할 수 있도록 제공합니다. 본 절에서는 MySQL 내장 알고리즘을 사용한 데이터베이스 암호화와 AVX2-PIPO16-FF1을 활용한 데이터베이스 암호화 결과를 비교 분석합니다. 실험 환경은 앞선 다른 성능 평가들과는 다르게 MySQL 환경 문제로 인해 2GB RAM Ubuntu 16.04 LTS 가상머신 상에서 수행되었습니다. 또한, 성능 비교 요소로는 암호화 후 데이터 저장 속도, 메모리 사용량을 선정하였습니다.

나-1) 암호화 후 데이터 저장 속도

표 5는 MySQL의 AES와 SHA2 그리고 본 구현물인 AVX2-PIPO-FF1를 사용하여 데이터를 암호화 한 후, 해당 데이터를 데이터베이스에 저장하기까지 걸리는 시간을 측정한 결과입니다. 10만 번의 암호화를 수행하여 10만 개의 데이터를 저장한 결과이며, 해당 결과를 반복 횟수인 10만으로 나눈 결과를 표 5로 작성하였습니다. 암호화할 데이터는 짧은 평문과 긴 평문을 임의로 지정하여 실험하였습니다.

표 5의 결과로 보아, AES와 SHA의 경우 패딩을 통해 블록길이를 맞추어 주므로 짧은 평문과 긴 평문에 대해서 비슷한 시간이 소요되는 것을 볼 수 있습니다. 또한, AVX2-PIPO16-FF1의 경우는 입출력 길이를 보존해주기 때문에 짧은 평문의 경우 더 짧은 시간에 암호화 및 데이터베이스로의 저장 과정을 수행하였습니다. 또한, 긴 평문이 입력될 경우 AES가 가장 빠른 연산 속도를 보였고, 다음으로 AVX2-PIPO16-FF1, SHA 순서로 빠른 속도를 나타내었습니다. SHA의 경우 평문이 256-bit로 가장 길기 때문에 이와 같은 결과가 나타난 것으로 보입니다. 정리하면, 짧은 평문에 대해서는 AVX2-PIPO16-FF1이 다른 두 알고리즘에 비해 각각 0.163ms, 0.226 ms 빨랐고, 긴 평문에 대해서는

AES가 다른 두 알고리즘에 비해 각각 0.118ms, 0.14ms 빠른 속도로 측정되었습니다. 또한, 표 5의 두 평문에 대한 결과를 종합하여 보면 AES는 0.7445ms, SHA는 0.846ms, AVX2-PIPO16-FF1은 0.722ms가 소요되었습니다.

실제로 데이터베이스에는 패스워드 등과 같이 길지 않은 데이터들이 주로 들어간다는 점과, 딥러닝 데이터 셋 등의 대규모 데이터베이스에 대한 암호화 필요, 데이터베이스에 저장되는 다양한 길이 및 형태의 데이터들에 대해 암호화를 수행해야하는 부분들을 고려하면, 데이터의 입출력 길이를 보존하여 평문 패딩 등의 과정을 수행하지 않아 작업속도를 높일 수 있는 AVX2-PIPO16-FF1을 사용하는 것도 속도 측면에서 효율적일 것으로 생각됩니다.

표 5. Comparison Result Table (Speed) : AES, SHA2 of MySQL and AVX2-PIPO16
(unit : ms)

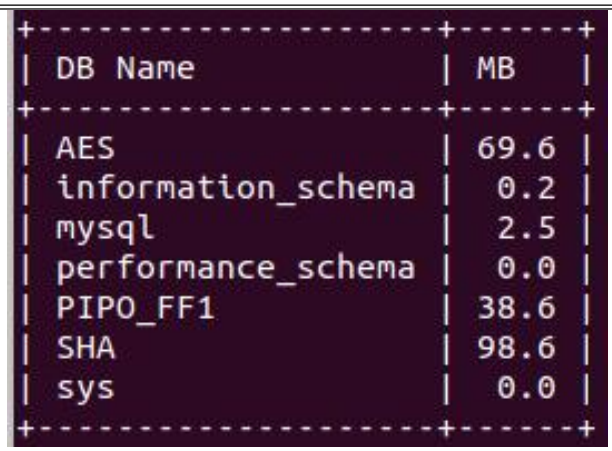
Data	AES	SHA2	AVX2-PIPO16-FF1
A1234	0.755	0.818	0.592
012345678901234567890 123456789012345678901 23456789ABCDEFGH	0.734	0.874	0.852

나-2) 메모리 사용량

그림 15은 표 5의 각 데이터를 10만 번 암호화하여 저장하는 것을 3번 반복한 후의 메모리 사용량입니다. 즉, 각 데이터베이스에 동일한 데이터를 암호화한 결과가 총 60만개 저장된 상태입니다. 메모리 용량은 SHA, AES, AVX2-PIPO16-FF1 순으로 더 많은 용량을 차지한 것을 확인할 수 있습니다. MySQL의 SHA의 경우 SHA2를 지원하므로 블록 길이가 256-bit이고, AES는 128-bit입니다. 따라서 평문의 길이가 해당 비트 수보다 적을 경우, 패딩을 수행하여 길이를 맞춰주기 때문에 입출력 길이가 동일한 형태보존암호가 적용된 AVX2-PIPO2-FF1보다 더 많은 메모리 공간을 차지할 수 밖에 없습니다. 실제로 저장된 결과는 그림 16의 (a), (b), (c) 에서 확인 가능하며, 블록 암호만 사용할 경우 고정된 길이로 변환된 후 저장됨을 알 수 있습니다.

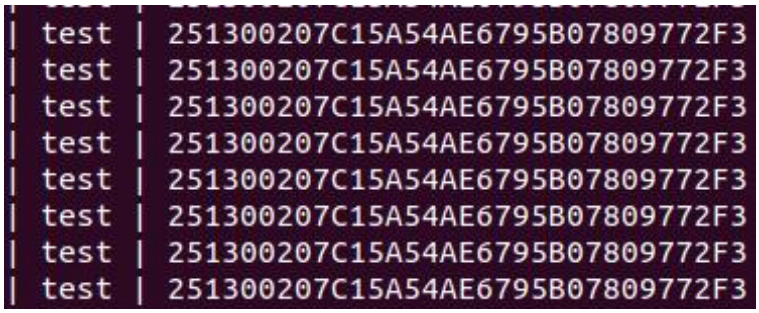
4.2.2 절의 가.에서 언급하였듯이 데이터베이스의 경우 저장되는 데이터의 도메인들이 컬럼 단위로 동일하며, 이러한 데이터베이스의 전체 스키마를 유지할 수 있도록 하여 시스템을 수정할 필요 없도록 하는 것이 중요한 부분이 됩니다. 이에 더불어, 각 데이터베이스가 차지하는 메모리 용량과 그에 따른 메모리 산정 및 시스템 유지 보수 또한 매우 중요한 요소입니다.

즉, 데이터 저장 시 사용되는 메모리 공간을 낭비하지 않아야 할 필요가 있으며, AVX2-PIPO16-FF1의 경우는 데이터의 입출력 길이가 보존되기 때문에 동일한 데이터를 저장하여도 암호 알고리즘의 블록 크기를 맞추기 위한 패딩으로 낭비되는 부분을 제거할 수 있습니다. 따라서 MySQL이 제공하는 기존의 AES, SHA2 보다 AVX2-PIPO16이 적용된 FF1을 통해 데이터 암호화를 수행하는 것이 더 효율적일 것으로 생각됩니다.



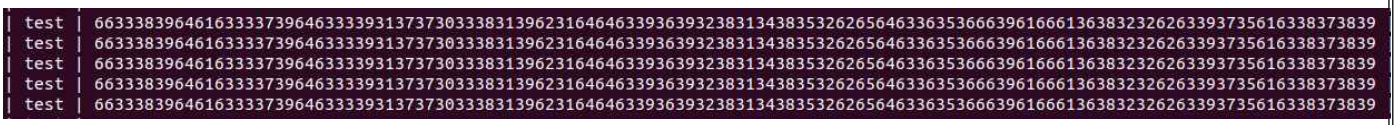
DB Name	MB
AES	69.6
information_schema	0.2
mysql	2.5
performance_schema	0.0
PIPO_FF1	38.6
SHA	98.6
sys	0.0

그림 15. 각 데이터베이스 메모리 사용량



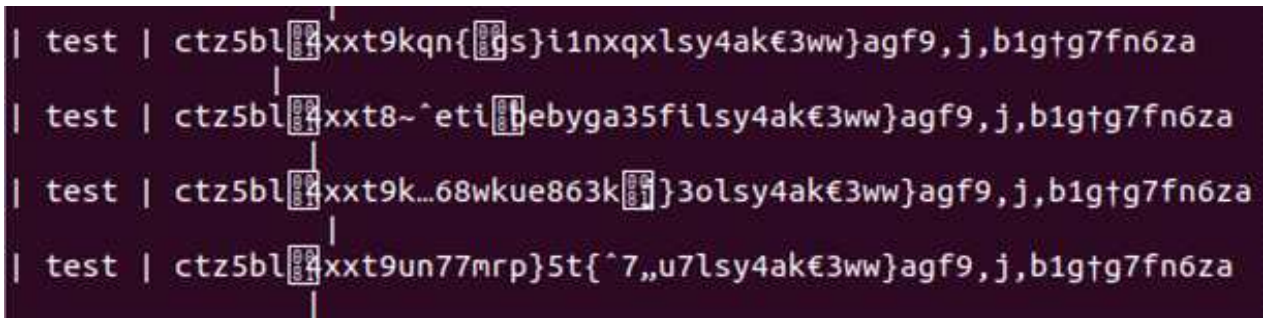
test	251300207C15A54AE6795B07809772F3
test	251300207C15A54AE6795B07809772F3
test	251300207C15A54AE6795B07809772F3
test	251300207C15A54AE6795B07809772F3
test	251300207C15A54AE6795B07809772F3
test	251300207C15A54AE6795B07809772F3
test	251300207C15A54AE6795B07809772F3
test	251300207C15A54AE6795B07809772F3

그림 16-(a) AES를 사용하여 암호화 된 후 저장된 데이터 (128-bit 고정)



test	6633383964616333373964633339313737303338313962316464633936393238313438353262656463363536663961666136383232626339373561633837383 9
test	6633383964616333373964633339313737303338313962316464633936393238313438353262656463363536663961666136383232626339373561633837383 9
test	6633383964616333373964633339313737303338313962316464633936393238313438353262656463363536663961666136383232626339373561633837383 9
test	6633383964616333373964633339313737303338313962316464633936393238313438353262656463363536663961666136383232626339373561633837383 9
test	6633383964616333373964633339313737303338313962316464633936393238313438353262656463363536663961666136383232626339373561633837383 9

그림 16-(b) SHA2를 사용하여 암호화 된 후 저장된 데이터 (256-bit 고정)



test	ctz5bl□4xxt9kqn{□gs}i1nxqxlsy4ak€3ww}agf9,j,b1g†g7fn6za
test	ctz5bl□4xxt8~^eti□bebyga35filsy4ak€3ww}agf9,j,b1g†g7fn6za
test	ctz5bl□4xxt9k…68wkue863k□}3olsy4ak€3ww}agf9,j,b1g†g7fn6za
test	ctz5bl□4xxt9un77mrp}5t{^7„u7lsy4ak€3ww}agf9,j,b1g†g7fn6za

그림 16-(c) AVX2-PIPO16-FF1을 사용하여 암호화 된 후 저장된 데이터 (입출력 길이 보존)

그림 16 MySQL의 AES, SHA2 그리고 AVX2-PIPO-FF1의 암호화 결과

4.3 GPU 연산 속도 비교

연산 속도 비교 결과는 그림 17과 같습니다. X축은 연산이 수행된 횟수이며, 각 횟수는 1부터 시작하여 10배씩 증가합니다. 곧, X축의 9는 100,000,000번의 연산이 수행되었을 때를 나타냅니다. Y축은 X축에 대해 연산 수행에 걸린 시간을 나타낸다. 걸린 시간의 단위는 ms입니다. 표 6는 CPU와 비교하여 GPU 연산 시간을 나타낸 결과이며, Improvement는 속도 향상 배율입니다.

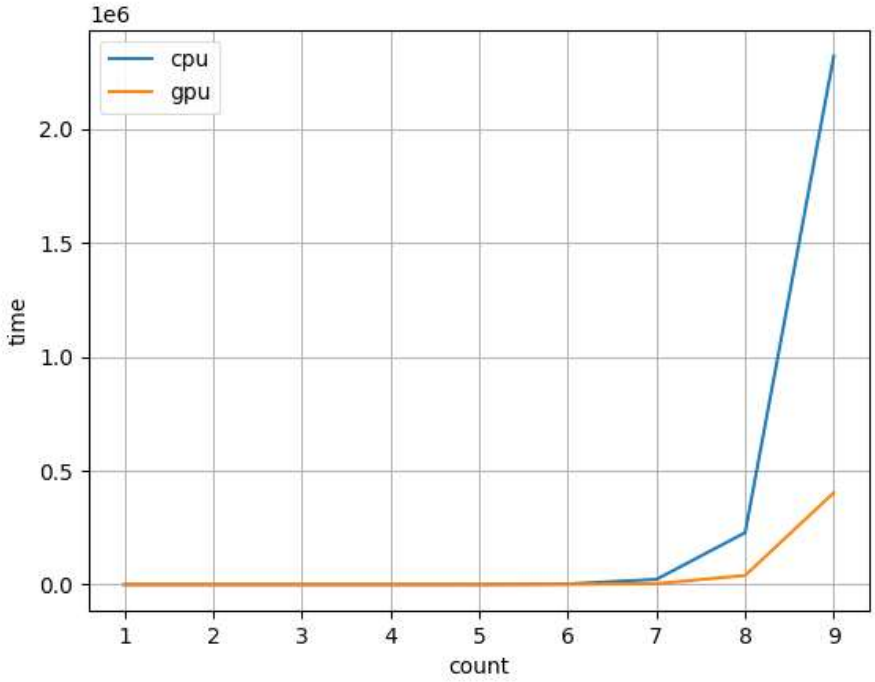


그림 17 Result of GPU calculation compare to CPU calculation as a graph

표 6 Result of GPU calculation compare to CPU calculation in detail
(unit : ms)

Count	CPU	GPU	Improvement
1	0	8	x0
10	0	3	x1
100	3	3	x1
1,000	24	7	x1
10,000	237	52	x5
100,000	2,303	421	x5.8
1,000,000	22,795	3,940	x5.575
10,000,000	227,686	39,798	x6.533
100,000,000	2,320,801	403,174	x6.837

연산 초기에는 CPU가 더 빠른 연산을 보여주었으나, 3번째 테스트인 100번의 연산부터 동일해졌으며 4번째 연산인 1,000번의 연산부터는 GPU의 연산 속도가 더 빨라지게 되어, 나중에는 5배 이상 빠른 결과를 보여주었습니다. 이를 통해 100번 이상의 연산이 수행될 경우에는 GPU를 이용하는 것이 더 효율적

이지만, 그 미만의 연산의 경우에는 CPU를 통해 연산을 수행하여도 충분하다는 결론을 내릴 수 있게 되었습니다.

또한, 그림 18와 같은 테스트를 통하여 GPU에서의 PIPO 구현을 형태보존암호에 적용하여 데이터베이스 암호화에 이용할 경우에 대한 예상치를 추정할 수 있습니다. GPU PIPO 병렬 구현은 기본적으로 2 사이즈의 데이터를 이용하며, 데이터베이스는 다량의 큰 사이즈의 데이터를 다룹니다. 그렇기에 GPU 병렬 연산에서도 속도에 대해 많은 향상이 있을 것으로 예상되며, 속도 향상의 결과는 결과 테이블 및 위 그래프에서의 속도 향상치와 비슷하게 데이터의 크기에 따라 선형적으로 증가할 것으로 예상됩니다.

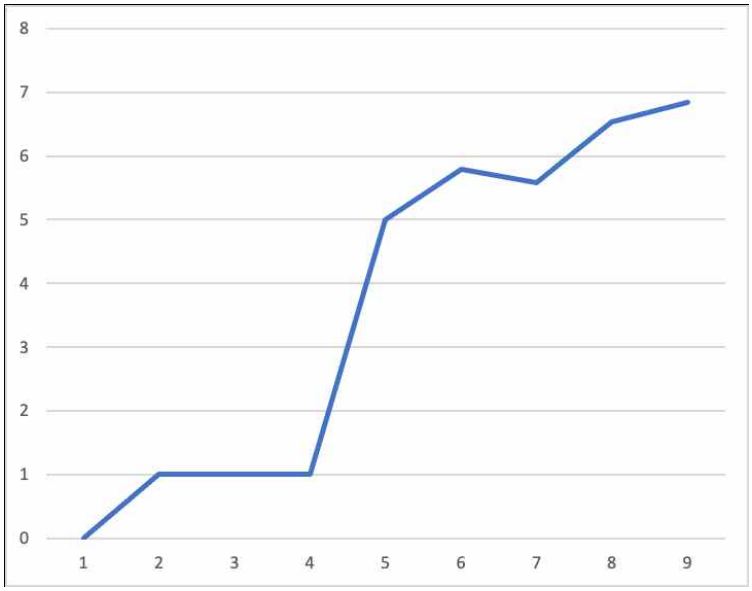


그림 18. Improvements of GPU calculation

5. 결론

경량 블록암호 PIPO를 AVX2와 GPU를 활용하여 병렬 구현하였습니다. AVX2의 경우 256-bit 레지스터를 활용하므로 PIPO의 경우는 32개의 평문 블록에 대한 연산을 동시에 수행할 수 있습니다. 그러나 AVX2가 8-bit Shift 명령어를 지원하지 않으므로 효율적인 16-bit 단위 Rotation 연산을 구현하기 위해 16개의 평문 블록을 동시에 연산하는 기법을 적용하였습니다. 병렬 구현된 AVX2-PIPO16과 AVX2-PIPO32의 cpb는 각각 기존 레퍼런스 코드와 비교하여 7.34배, 1.8배 향상되었음을 확인하였습니다.

다음으로, 병렬 구현된 AVX2-PIPO16을 형태보존암호인 FF1의 라운드 함수에 적용하였습니다. 형태보존암호 중 한 종류인 FF1의 경우 라운드 함수의 PRF와 CIPH에 사용되는 블록암호로 AES-128을 사용합니다. 해당 부분에 AVX2-PIPO16을 적용하여 내부적으로 128 바이트의 평문을 동시에 처리할 수 있도록 하였습니다.

또한, 형태보존암호는 그 특징 때문에 주로 데이터베이스 암호화에 사용되기 때문에 AVX2-PIPO16-FF1을 데이터베이스 암호화에 응용해보았습니다. 형태보존암호를 통해 데이터를 암호화할 때, 그 형식과 길이가 보존된다는 점은 데이터베이스의 효율적인 활용 측면에서 이점이 됩니다.

성능 측정 결과, 기존 MySQL에서 지원하는 AES, SHA2 보다 임의의 평문 길이에 대해 평균적으로 더

빠른 속도 (0.722ms 소요되었으며, AES: 0.7445 SHA : 0.846ms)를 보였습니다. 그리고, 암호화 할 데이터의 길이가 짧아질 경우, 패딩이 필요한 다른 블록암호 알고리즘들과는 다르게 짧은 평문 그대로를 암호화하기 때문에 AES와 SHA에 비해 각각 0.163ms, 0.226 ms 만큼 빠른 암호화 및 데이터베이스 저장 속도를 달성하였습니다. 또한 메모리 용량 관점에서도 AVX2-PIPO16-FF1이 다른 두 블록암호에 비해 월등한 성능을 보였습니다. 블록 길이에 맞추기 위한 패딩 과정이 필요하지 않으므로 데이터 저장 시 메모리 공간 낭비를 줄일 수 있습니다.

대규모 데이터베이스에 대한 암호화 필요, 다양한 길이 및 형태의 데이터들에 대한 암호화 등을 고려하면, 작업속도 및 메모리 사용량 관점에서 더 좋은 성능을 얻을 수 있는 AVX2-PIPO16-FF1을 사용하는 것이 효율적일 것이라는 결론을 내렸습니다.

또한, GPU를 통한 병렬 처리는 기존의 PIPO 처리속도에 비해 최대 약 7배에 달하는 높은 성능을 보여줬습니다. CPU에서 직렬로 처리하던 데이터를 쪼개어 블록 단위를 통해 대량의 데이터를 한 번에 처리하며, 이를 통해 병렬 처리된 PIPO가 다양한 분야에 응용될 수 있다는 가능성을 보여주었습니다. NVIDIA의 쿠다를 이용한 이 병렬 처리는 다량의 데이터를 빠르게 받아들일 수 있으며 높은 응용성을 보이기 때문에 향후에도 더 다양한 분야에서 성능이 극대화 된 암호화를 할 수 있을 것으로 예상됩니다.

AVX2-PIPO32의 경우 최적화 옵션을 높게 설정하면 속도가 더욱 빨라짐을 확인하여 AVX2-PIPO16 또한 컴파일 옵션을 -O2로 설정해보았으나, 시간이 측정되지 않는 오류가 생겨 -O1으로 측정하였습니다. 해당 부분을 해결하면 성능 향상이 조금 더 있을 것으로 생각됩니다.

향후 연구로는 병렬 구현된 PIPO와 형태보존암호의 장점을 모두 활용할 수 있는 이미지 부분 암호화에 대한 부분을 진행할 계획입니다. 또한, GPU를 활용한 PIPO 병렬 구현물도 형태보존암호 FF1에 적용할 것이며, 이를 데이터베이스 암호화 및 이미지 부분 암호화에도 적용하여 대규모 데이터베이스 및 이미지에 대한 효율적인 암호화 방안에 대해 연구를 진행할 계획입니다.