CUDA GPU 상에서
경량 블록 암호 구현에 대한 고찰

https://youtu.be/qIpTkAsp6H8

GPU 상에서 암호 구현

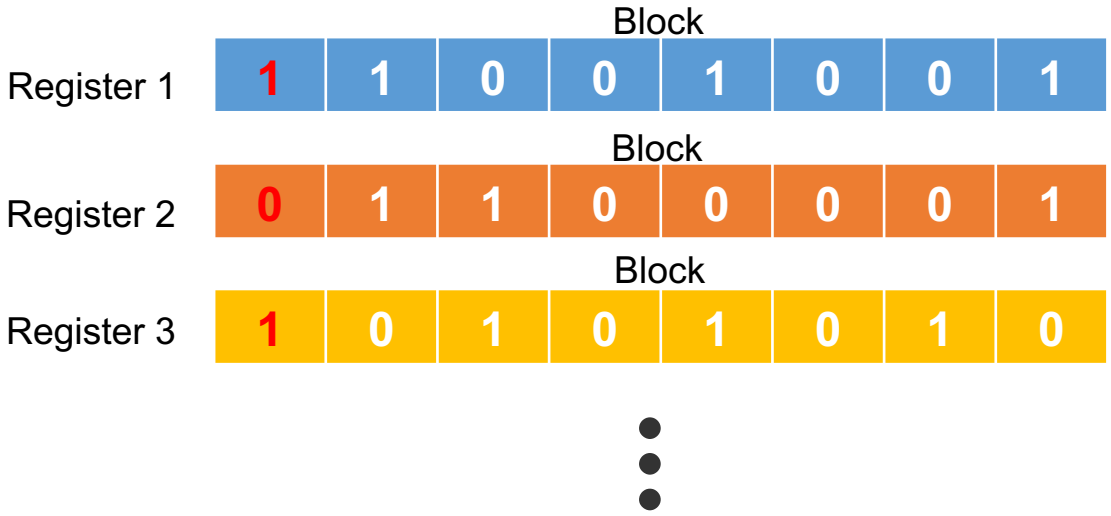
- 사물인터넷(IoT), 클라우드 컴퓨팅, 빅데이터 등의 확산, 애플리케이션에 대한 **고속 암호화** 의 필요성이 대두
- 최신 **그래픽 프로세서**는 **높은 처리 능력**을 제공 **암호 처리**에 그래픽 프로세서를 활용 연구 진행 중
  - GPU는 **파일 또는 전체 디스크 암호화**를 위한 암호화 보조 프로세서로 사용되어 CPU 암호화로 인한 성능 손실을 제거
  - CPU 사용량이 많은 SSL **서버에서 GPU를 보조 프로세서**로 사용하면 암호화 부담에서 벗어 나 다른 작업에 CPU 성능을 사용 가능
  - GPU 최적화는 GPU가 이론적으로 얻은 **암호 분석 결과 또는 축소된 버전을 합리적인 시간에 검증**하는 데 사용
- 경량 블록암호 PRESNET, GIFT, **PIPO**를 CUDA GPU 상의 최적 구현 하였음
  - **비트슬라이싱 기법**을 적용한 최적화, GPU 요소를 최대한 사용하여 높은 처리량 달성
- 공통된 특징을 기반으로 CUDA GPU 상에서 경량 블록 암호 구현에 대하여 고찰함

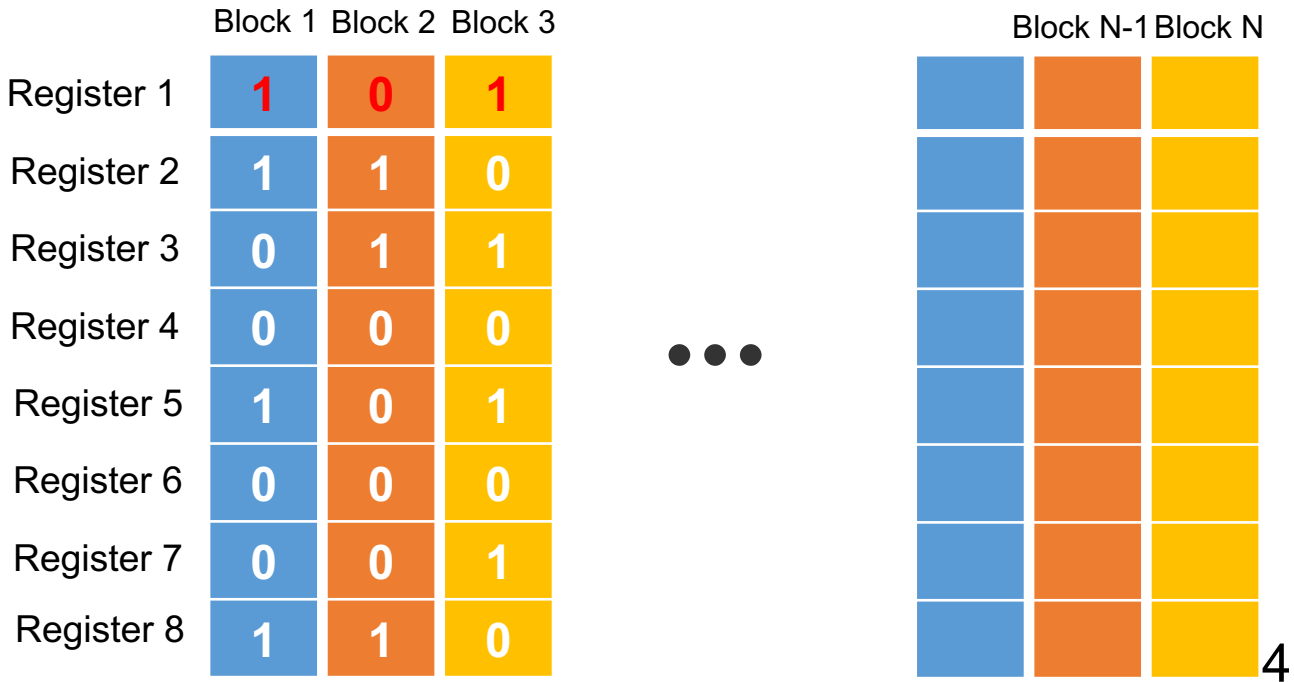
적용 기법들

- 비트슬라이싱
- 공유 메모리
- 커널 내부 루프
- 블록 당 스레드(스레드 수), 그리드 당 블록(블록 수) 조절

Bitslicing

- **Eli Biham**이 A Fast New **DES** Implementation in Software 라는 논문에서 처음 사용
- 비트슬라이싱은 여러 블록의 비트들을 각 순서대로 모아 **비트 단위로 연산**하는 기법
- 여러 블록을 병렬로 처리 가능
- 하드웨어에서 논리 회로를 구현하는 것처럼 **함수을 단일 비트 논리 연산**(*AND* , *XOR* , *OR* , *NOT* 등)로 표현
- 여러 블록을 비트 슬라이싱 표현으로 바꾸기 위한 **packing 과정**, 다시 원래 형태의 블록으로 되돌리기 위한 **unpacking 과정** 추가되며 이러한 **오버헤드도 고려** 필요





Bitslicing implementation on Software

- Bitslicing기법으로 하드웨어 지향 블록암호 PRESENT, GIFT을 소프트웨어에서 효율적으로 구현
  - PRESENT는 4비트 S-box와 64비트 순열로 구성되어 있어 효율적인 소프트웨어에서 구현은 비효율적
  - Reis et al.*가 조회 테이블 대신 새로운 순열, 최적화된 부울 공식, S-box의 비트 슬라이싱 구현을 사용하여 기존 Cortex-M3에서 최상의 어셈블리 구현을 이전 작업보다 8배 향상
  - Adomnicai et al.**가 GIFT을 Fix-slicing이라는 새로운 기술로 몇 번의 회전만 사용하여 매우 효율적인 소프트웨어 구현 당시 최고의 기존 Cortex-M3에서 AES 일정 시간 구현보다 빠른 성능을 보임
- Hajihassani et al.은 비트 슬라이스 기술을 사용하여 V100에서 1,478Gbps를 달성하는 가장 빠른 AES-128 구현을 제시
  - 컴파일하는 동안 라운드 키를 하드 코딩 조건 필요
- 조회 테이블 대신 새로운 순열, 최적화된 부울 공식 사용

Bitslicing

- 여러 블록을 병렬로 처리하므로 대량의 데이터처리에 적합
  - GPU에서의 암호화 목적과 동일
  - 더 높은 처리량 달성 하였음

- Packing, Unpacking 과정 추가
  - 추가적인 오버헤드 발생하므로 Packing, Unpacking이 단순할 수록 효율
  - 키 탐색 공격에는 작은 키 사이즈의 암호가 효율적
  - 대량 데이터 암호화에서는 작은 블록 사이즈의 암호가 효율적

- 레지스터 사용 증가
  - CUDA에서는 일반적으로 스레드 수, 블록 수가 많을수록 좋은 성능 보임
  - 스레드 수, 블록 수가 많을수록 사용되는 레지스터는 많지만 레지스터양은 제한됨

- GPU 구현에 Bitslicing기법을 적용은 경량암호의 대량 처리와 키 탐색에 적합
  - 단순한 키 스케줄, 작은 키 사이즈, 작은 블록 사이즈

실험

- 구현 성능 테이블 vs Bitslicing

- Packing, Unpacking 과정의 오버헤드

PRESENT : Table base vs Bitslicing

- 키 탐색
  - Cihangir Tezcan, Key lengths revisited: GPU-based brute force cryptanalysis of DES, 3DES, and PRESENT, Journal of Systems Architecture, Volume 124, 2022
  - 커널 내부에서 key 값 1씩 증가

Table 2
The results of our PRESENT exhaustive search attacks on different GPUs.

GPU	PRESENT Exhaustive Search
MX 250	$116,197,180 \approx 2^{26.79}$ keys/s
GTX 860M	$121,949,098 \approx 2^{26.86}$ keys/s
Tesla k20	$340,904,013 \approx 2^{28.34}$ keys/s
GTX 970	$377,758,044 \approx 2^{28.49}$ keys/s
RTX 2070 S	$887,917,367 \approx 2^{29.73}$ keys/s
RTX 3070	$1,885,204,563 \approx 2^{30.81}$ keys/s

- Table base
  - RTX 3070 1.8852 Gigakeys/s
- Bitslicing
  - Packing 상태에서 1씩 증가 구현, 암호문 Unpacking 수행후 비교
  - RTX 3060 7.8917 Gigakeys/s
- 4.18배 차이
- Bitslicing 구현이 높은 처리량 달성

Packing, Unpacking

- 64bit, 128bit 평문 Packing, Unpacking 오버헤드 확인
- GIFT-64, GIFT-128 구현으로 확인

Packing, Unpacking	X	O	오버헤드 비율
GIFT-64	736.7403 Mh/s	641.2404 Mh/s	13%
GIFT-128	314.9770 Mh/s	193.1445 Mh/s	39%

- Bitslicing 구현에서
  대량 데이터 암호화에서는 작은 블록 사이즈의 암호가 효율적

키 탐색 vs 대량 데이터 처리

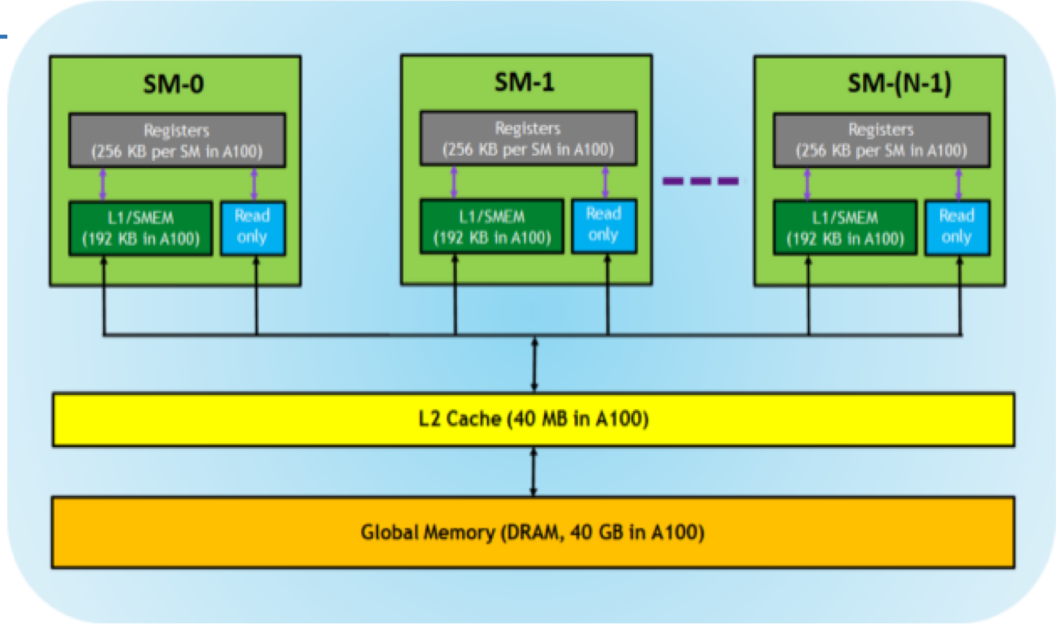
- 키 전수 조사의 경우 GPU로 데이터 전송을 최소화 할 수 있음
  - 10배 이상 차이

데이터 전송	O	X
속도	604.8 Mkeys/s	7891.7 Mkeys/s

공유 메모리

- Shared memory
  공유 메모리는 GPU에서 가장 빠른 메모리

- 뱅크라는 32개의 동일한 사이즈 모듈로
  나뉘며 각각 뱅크는 동시에 액세스 가능



- 워프는 32개의 스레드 묶음,
  워프의 각 스레드가 같은 뱅크에 액세스하면 뱅크 충돌이 발생

- AES 의 T 테이블을 공유 메모리에 보관하면 속도가 최소 10.23배 빨라짐*
  - GPU 커널에서 각 스레드는 전역 메모리에서 복사 후 공유 메모리에 저장
  - **뱅크 충돌이 발생하지 않도록**
    T[0], T[31], T[63], T[95], T[127], T[159], T[191], T[223]를 뱅크 0
    T[1], T[32], T[64], T[96], T[128], T[160], T[192], T[224]를 뱅크 1

실험

- 여러 논문에서 테이블 구현 시 공유 메모리 사용, 뱅크 충돌 방지 적용

➢Bitslicing 구현 공유 메모리 사용?

- 공유 메모리 동적 할당 적용, PRESENT 80 비트슬라이싱 구현에서 비교
- 공유 메모리 사용 시 다소 성능향상

공유 메모리 사용	X	O	성능향상
속도	604.8 Mkeys/s	617.2 mh/s	3%

커널 내부 루프

- GPU 레지스터로 평문(키)를 로드 하고 커널 내부에서 (제한된)수정으로 루프
  - GPU로 전송하는 데이터 최소화
  - 메모리 액세스를 최소화

Ex) 평문(키)의 일부만 변경, 키 전수 조사

A2	18	04	E0	?	?	?	?

- 고정되는 A2 18 04 E0 61 78만 GPU로 전송
  한번만 메모리 액세스하여 레지스터 저장 이후 for 문으로 1씩 증가
- 키 : 0, 1 ,2 ,3 …

Ex) 테이블 구현에서 호출 후 for 문에서 반복 사용

실험

- Bitslicing 내부 루프 사용
- PRESENT key 탐색
- Ex) 그리드당 블록 *4, 내부 for 문 *4
- 차이 X

블록 당 스레드, 그리드 당 블록 조절

- 많으면 많을 수록 좋았음

- 제한 있음
  - 어떤 요인인지 확인 필요

Q & A