情報保護學會論文誌
第 22 卷 第 3 號, 2012. 6

HIGHT 블록 암호 알고리즘의 고속화 구현*

백 은 태,† 이 문 규‡
인하대학교 컴퓨터정보공학부

Speed-optimized Implementation of HIGHT Block Cipher Algorithm*

Eun-Tae Baek,† Mun-Kyu Lee‡
School of Computer and Information Engineering, Inha University

요 약

본 논문에서는 국제 표준 블록 암호 알고리즘인 HIGHT를 CPU 및 GPU 상에서 소프트웨어로 고속화 구현하기 위한 다양한 방법을 시도한다. 먼저 CPU 상에서는 32비트 및 64비트 운영체제를 고려하고 비트 슬라이싱 및 바이트 슬라이싱 기법을 적용한다. 이들 최적화 기법의 적용 결과, Intel core i7 920 CPU 상에서 64비트 운영체제를 이용할 경우 최대 1.48Gbps의 속도를 보여 슬라이싱이 적용되지 않은 기존 구현에 비해 최대 2.4배 빠른 성능을 확인할 수 있었다. 한편 GPU 상에서는 NVIDIA의 CUDA 라이브러리를 활용하였으며, 서브키 및 F 함수를 위한 룩업 테이블 등과 같이 자주 사용되는 데이터를 공유 메모리에 저장하여 사용하고, 전역 메모리에서 데이터를 읽어올 때는 통합 접근(coalesced access) 기법을 사용하는 등 최적화 기법들을 적용해 구현하였다. 특히 본 논문은 GPU 상에서 HIGHT를 최적화한 최초의 결과로, GPU 상에서도 바이트 슬라이싱 기법을 적용할 경우 단순 구현 결과보다 20% 이상 빠른 성능을 확인할 수 있었으며, CPU에 비해서는 약 31배 빠른 결과를 얻을 수 있었다.

ABSTRACT

This paper presents various speed optimization techniques for software implementation of the HIGHT block cipher on CPUs and GPUs. We considered 32-bit and 64-bit operating systems for CPU implementations. After we applied the bit-slicing and byte-slicing techniques to HIGHT, the encryption speed recorded 1.48Gbps over the intel core i7 920 CPU with a 64-bit operating system, which is up to 2.4 times faster than the previous implementation. We also implemented HIGHT on an NVIDIA GPU equipped with CUDA, and applied various optimization techniques, such as storing most frequently used data like subkeys and the F lookup table in the shared memory; and using coalesced access when reading data from the global memory. To our knowledge, this is the first result that implements and optimizes HIGHT on a GPU. We verified that the byte-slicing technique guarantees a speed-up of more than 20%, resulting a speed which is 31 times faster than that on a CPU.

Keywords: HIGHT, Block Cipher, Bit-slice, Byte-slice, GPGPU, CUDA

접수일(2011년 10월 17일), 게재확정일(2011년 12월 10일)
* 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단-차세대정보컴퓨팅기술개발사업의 지원을 받아 수행된 연구임(No. 2011-0029926)
† 주저자, euntae@hotmail.com
‡ 교신저자, mklee@inha.ac.kr

I. 서 론

HIGHT (HIGh security and light weight)[1][2]는 RFID나 센서 네트워크 노드 등과 같이 저전력, 경량화를 요구하는 컴퓨팅 환경에서 기밀성을 제공하기 위한 목적으로 2005년 개발된 국산 64비트 블록 암호 알고리즘이다.

HIGHT는 이미 2010년 ISO/IEC 국제표준[3]으로 채택되는 등 그 실효성이 검증되었음에도 불구하고, DES[4] 및 AES[5] 등의 표준 암호는 물론 국제 표준으로 제정되어 있는 다른 국산 알고리즘인 SEED[6]와 국내 표준인 ARIA[7]에 비해서도 최적화에 대한 연구가 다양하게 이루어지지 않고 있다. 특히 CUDA 라이브러리를 이용한 GPU 환경에서의 블록 암호 알고리즘 구현이 이미 DES, AES, SEED, ARIA 등[8][9][10][11]에서 다양하게 이루어져 온 반면에, HIGHT에 대해서는 이러한 결과가 발표된 바 없다. HIGHT가 비록 저전력, 경량화를 요구하는 컴퓨팅 환경을 위한 암호 알고리즘이지만 암호분석관점에서 또는 HIGHT를 사용하는 센서의 데이터를 다량으로 처리하는 중앙 서버 등에서는 HIGHT의 소프트웨어 최적화 기술이 반드시 필요하다. 이에 본 논문에서는 CPU와 GPU 환경에서 HIGHT의 소프트웨어 구현을 고속화하기 위한 다양한 방법을 시도하고 성능을 평가한다. 본 논문은 HIGHT의 GPU 구현을 시도한 최초 결과이다.

본 논문에서는 우선 비트 슬라이싱(bit-slicing) 및 바이트 슬라이싱(byte-slicing) 기법을 이용하였다. 비트 슬라이싱 기법은 Eli Biham에 의해 소개되었으며[12], DES, AES 등의 주요 암호 알고리즘 구현에 적용된 바 있다[12][13]. 비트 슬라이싱 기법은 레지스터의 길이가 길수록, 레지스터의 개수가 많을수록, 대상 알고리즘이 하드웨어로 작게 구현 가능할수록 효율적으로 이용할 수 있으며, HIGHT의 경우 비교적 간단한 구조로 이루어져 있기 때문에 이 기법을 적용하여 효율적으로 구현이 가능하다. 바이트 슬라이싱은 같은 순서의 바이트들을 묶어서 한 번의 수행으로 여러 블록이 동시에 수행되는 것과 같은 효과를 보여주는 기법이며, HIGHT와 같이 바이트 단위로 연산되는 알고리즘에 적합하다. 본 논문에서는 CPU 상에서 비트 슬라이싱 및 바이트 슬라이싱 기법을 모두 구현하여 기존 구현과 비교하였다.

한편 GPU에서는 복잡한 쓰레드 및 메모리 계층 구조로 인해 최적화에 고려할 요소가 더 많은데, 본 논문에서는 통합 접근(coalesced access) 기법을 이용하여 캐시 히트 비율을 높이고 각종 참조 테이블과 서브키 등 자주 활용되는 데이터는 가능하면 온 칩 메모리를 사용하도록 하여 데이터 접근 시간을 최소화하였다. 또한 여러 쓰레드에서 공통적으로 사용되는 데이터를 전역 메모리에서 공유 메모리로 복사할 때 각 쓰레드가 오버헤드를 병렬적으로 분담할 수 있도록 멀티 프로세서당 쓰레드 개수를 최적화하였다. 구현 기법 측면에서는 GPU는 레지스터 크기가 32비트로 작고 개수도 제한적이어서 비트 슬라이싱 구현에 적합하지 않으므로 GPU 상에서는 바이트 슬라이싱 기법만을 고려하였다.

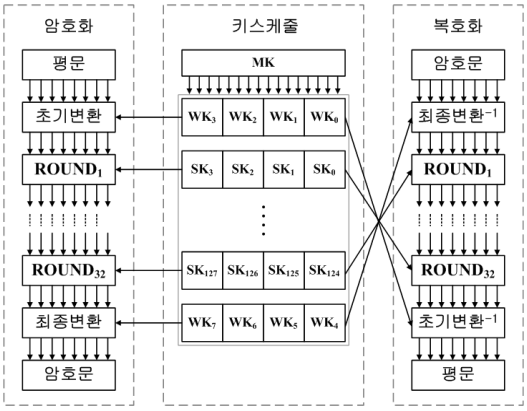
본 논문의 구현 및 실험 결과, 64비트 CPU인 Intel core i7 920에서 비트 슬라이싱은 1.48Gbps, 바이트 슬라이싱은 0.98Gbps의 속도로, 한국인터넷진흥원(Korea Internet & Security Agency; 이하 KISA)에서 제공해주는 소스코드[1]를 그대로 실행했을 때의 결과인 0.62Gbps보다 높은 성능을 얻을 수 있었다. GPU에서는 NVIDIA사에서 제공해주는 CUDA 라이브러리[14]를 이용하여 구현하였는데, KISA에서 제공하는 CPU용 소스코드를 NVIDIA GeForce GTX 470 GPU에 포팅했을 때 38Gbps의 성능을 보인 데 비해, 바이트 슬라이싱을 추가로 적용할 경우 46Gbps로 더 좋은 성능을 얻을 수 있었다. CPU와 GPU 간의 성능을 비교하면 약 31배 빠름을 확인할 수 있다.

본 논문은 다음과 같이 구성된다. 2장에서는 HIGHT 알고리즘에 대하여 서술하고, 3장에서는 비트 슬라이싱 기법과 바이트 슬라이싱 기법을 이용한 CPU 상에서의 구현, 4장에서는 NVIDIA CUDA 라이브러리를 이용한 GPU 상에서의 구현에 대하여 설명한다. 5장에서는 실험 결과에 대해 서술하고, 마지막으로 6장에서 결론을 맺는다.

II. HIGHT

HIGHT 블록 암호[1][2]는 2005년 KISA, 구) 국가보안연구소 및 고려대가 공동으로 개발한 64비트 블록암호 알고리즘이며, 2010년 ISO/IEC 국제표준으로 채택되었다[3]. HIGHT는 128비트 키, 64비트 평문으로부터 64비트 암호문을 출력한다. 제한적 자

1) KISA에서는 웹사이트[1]의 'HIGHT 보급신청'을 통해 HIGHT 소스코드를 배포하고 있음



〔그림 1〕 HIGHT의 전체 구조〔1〕

원을 갖는 환경에서 구현될 수 있도록 8비트 단위의 기본적인 산술 연산들인 XOR, 덧셈, 순환 이동만으로 설계되었다. HIGHT는 128비트 키를 입력으로 받아 8바이트의 화이트닝키와 128바이트의 서브키를 생성하여 초기변환과 각 라운드 연산에 사용한다. HIGHT는 〔그림 1〕에서 볼 수 있듯이 화이트닝키를 가지고 초기변환을 한 후 서브키를 이용하여 32개의 라운드 연산을 한다. 그리고 마지막 최종변환을 한 후 암호문을 얻는다. HIGHT는 Feistel 구조로 이루어져 있으므로 복호화도 암호화와 유사한 연산으로 구성되어 있다.

HIGHT는 덧셈, XOR, 순환 이동 연산으로 이루어져 있으며, 덧셈은 $A+B \bmod 2^8$ 형식으로 일반적인 덧셈을 한 후 모듈러(modular) 연산을 취하는 형태를 가지며 본 논문에서는 $+$로 나타낸다.

XOR는 $\oplus$로 나타내고 순환 이동은 $\lll i$와 같이 나타낸다. 64비트의 평문과 암호문은 각각 8개의 바이트로 다음과 같이 나타낸다.

$$P = P_7 \| P_6 \| \cdots \| P_0$$

$$C = C_7 \| C_6 \| \cdots \| C_0$$

64비트 라운드 함수 입출력은 8개의 바이트로 다음과 같이 나타내고 i는 몇 번째 라운드인지를 가리킨다.

$$X_i = X_{i,7} \| X_{i,6} \| \cdots \| X_{i,0}\ ,\ i = 0,\cdots,32$$

라운드 함수에 적용되는 라운드키는 서브키와 화이트닝키로 각각 $SK_i\ ,\ i=0,\cdots,127$, $WK_i\ ,\ i=0,\cdots,7$와 같이 나타낸다.

HIGHT의 초기변환은 화이트닝키와 함께 다음과 같은 연산으로 이루어진다.

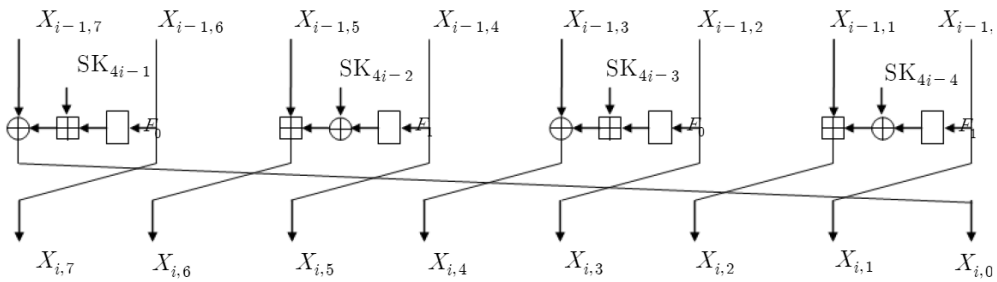
초기변환:

$$X_{0,i} = P_i, i = 1,3,5,7$$
$$X_{0,0} = P_0 + WK_0$$
$$X_{0,2} = P_2 \oplus WK_1$$
$$X_{0,4} = P_4 + WK_2$$
$$X_{0,6} = P_6 \oplus WK_3$$

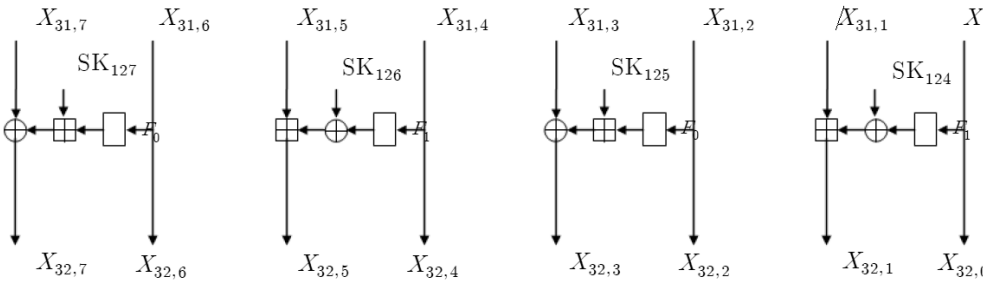
HIGHT의 라운드 함수는 다음과 같은 두개의 보조 함수를 갖는다. KISA에서 제공해주는 HIGHT 소스코드는 다음의 보조 함수의 입력 및 출력이 1 바이트로 표시될 수 있다는 특성을 활용하여 미리 계산된 테이블 형태로 저장하여 구현하고 있다.

$$F_0(X) = X^{\lll 1} \oplus X^{\lll 2} \oplus X^{\lll 7}$$
$$F_1(X) = X^{\lll 3} \oplus X^{\lll 4} \oplus X^{\lll 6}$$

라운드 함수는 Round_1부터 Round_{32}까지 32회 반복한다. 각 라운드 함수 Round_i, $i = 1,\cdots,31$ 은 〔그림 2〕와 같으며, 다음과 같이 계산된다.



〔그림 2〕 HIGHT 라운드 함수〔1〕



〔그림 3〕 HIGHT의 마지막 라운드〔1〕

$$X_{i,j} = X_{i-1,j-1}, j = 1,3,5,7$$
$$X_{i,0} = X_{i-1,7} \oplus (F_0(X_{i-1,6}) + SK_{4i-1})$$
$$X_{i,2} = X_{i-1,1} + (F_1(X_{i-1,0}) \oplus SK_{4i-4})$$
$$X_{i,4} = X_{i-1,3} \oplus (F_0(X_{i-1,2}) + SK_{4i-3})$$
$$X_{i,6} = X_{i-1,5} + (F_1(X_{i-1,4}) \oplus SK_{4i-2})$$

마지막 라운드는 바이트들을 섞지 않는다. 따라서 〔그림 3〕과 같고, 다음과 같이 계산된다.

$$X_{32,i} = X_{31,i}, i = 0,2,4,6$$
$$X_{32,1} = X_{31,1} + (F_1(X_{31,0}) \oplus SK_{124})$$
$$X_{32,3} = X_{31,3} \oplus (F_0(X_{31,2}) + SK_{125})$$
$$X_{32,5} = X_{31,5} + (F_1(X_{31,4}) \oplus SK_{126})$$
$$X_{32,7} = X_{31,7} \oplus (F_0(X_{31,6}) + SK_{127})$$

HIGHT의 최종변환은 다음과 같이 계산된다.

최종변환:

$$C_i = X_{32,i}, i = 1,3,5,7$$
$$C_0 = X_{32,0} + WK_4$$
$$C_2 = X_{32,2} \oplus WK_5$$
$$C_4 = X_{32,4} + WK_6$$
$$C_6 = X_{32,6} \oplus WK_7$$

III. CPU 상의 HIGHT 고속화 구현

본장에서는 CPU 환경에서의 HIGHT 고속화 구현에 대해서 다룬다. 본 논문에서는 비트 슬라이싱 기법과 바이트 슬라이싱 기법을 적용하여 구현하였으며, 아래에서는 이들에 대해 각각 설명한다.

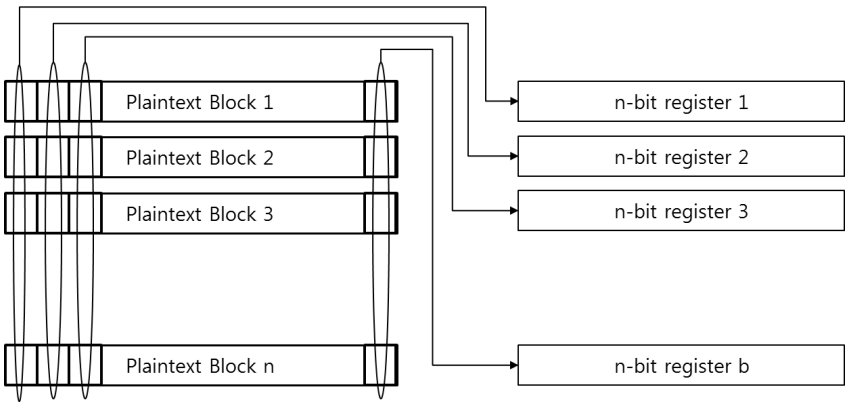
3.1 비트 슬라이싱 구현

비트 슬라이싱 기법은 1997년 Eli Biham에 의해 제안되었다〔12〕. 이 기법은 〔그림 4〕에서 볼 수 있는 것과 같이 여러 블록들의 비트들을 각 순서대로 모아 비트 단위의 연산을 하는 방법이다. 즉, 각 블록의 첫 번째 비트들을 모아 한 개의 데이터를 형성하고, 두 번째 비트들을 모아 또 한 개의 데이터를 형성하고, 이와 같은 방법으로 n번째 비트까지 데이터를 형성한다. 그리고 이와 같이 모아진 데이터를 입력으로 받아 하나의 명령어로 한꺼번에 연산을 수행한다. n비트 레지스터 환경에서 이 연산이 진행된다면 n개의 블록이 동시에 처리되는 것과 같은 효과를 볼 수 있다.

비트 슬라이싱 기법은 비트 단위의 연산으로 이루어지기 때문에 하드웨어 구현과 유사하고, AND, OR, XOR, NOT 등과 같은 간단한 논리 게이트들로 구현된다. 레지스터의 크기에 따라 동시에 수행할 수 있는 블록의 개수가 결정되므로 레지스터의 크기가 클수록 유리하고, 레지스터 수가 적으면 일부 데이터를 메모리에 저장하는 오버헤드가 발생하므로 레지스터의 개수가 많을수록 유리하다. 또한 비트 슬라이싱 구현은 하드웨어 구현과 구조가 유사하므로 대상 알고리즘을 하드웨어로 작게 구현 가능할수록 이 방법을 이용하여 효율적인 구현을 할 수 있다.

HIGHT는 XOR, 덧셈, 순환 이동 등의 간단한 연산들로 이루어져 있기 때문에 비트 슬라이싱을 적용하면 효율적으로 구현이 가능하다. 본 논문에서는 32비트 OS 환경의 CPU와 64비트 OS 환경의 CPU로 나눠 비트 슬라이싱 기반으로 HIGHT를 구현하였다. 32비트 OS 환경의 CPU에서는 사용하는 레지스터의 크기가 32비트 이고, 64비트 OS 환경의 CPU에서는 레지스터의 크기가 64비트이다. 연산 부분에 대한 구현은 32비트와 64비트 모두 같은 방식으로 구현하였다.

XOR의 경우에는 특별한 조작 없이 비트들을 병렬로 모아서 수행하면 된다. 다만 비트별로 데이터가 형



〔그림 4〕 비트 슬라이싱 데이터 구조

성되어 있기 때문에 바이트 단위의 연산을 하는 HIGHT의 경우 한 바이트를 처리하기 위해 8 번의 XOR가 필요하다. 1번의 XOR 연산이 8번으로 늘어났지만 32비트 환경에서는 32개의 블록이, 64비트 환경에서는 64개의 블록이 동시에 연산이 되는 것과 같으므로 실제로는 XOR를 위한 블록 당 연산 횟수가 각각 0.25번, 0.125번으로 아주 적다.

덧셈 연산의 경우에는 비트별로 연산을 진행하기 위해 8비트 전가산기를 위한 에뮬레이터를 구현하였다. 기존 구현 방법으로 구현했을 때는 한 번의 덧셈 연산으로 간단하게 구현이 되지만 비트 단위의 연산에서는 캐리 값을 처리하기 위해서 다소 복잡해진다. 에뮬레이터에 대한 분석 결과 덧셈 연산의 비트 슬라이싱 구현은 34개의 연산으로 이루어진다. 하지만 덧셈 연산 역시 블록 한 개당 실제 연산 횟수를 계산해보면 32비트 환경에서는 1.06번, 64비트 환경에서는 0.53번으로 기존 연산과 비슷하거나 더 빠른 성능을 보여준다.

순환 이동 연산의 경우는 원하는 위치에 값을 넣어주기만 하면 되기 때문에 아주 비용이 적고, 값을 넣어주는 연산을 하지 않고 다음 연산을 할 때 바뀌는 위치를 찾아 그에 맞게 바로 대입을 해주면 연산을 할 때 비용이 전혀 들지 않는다.

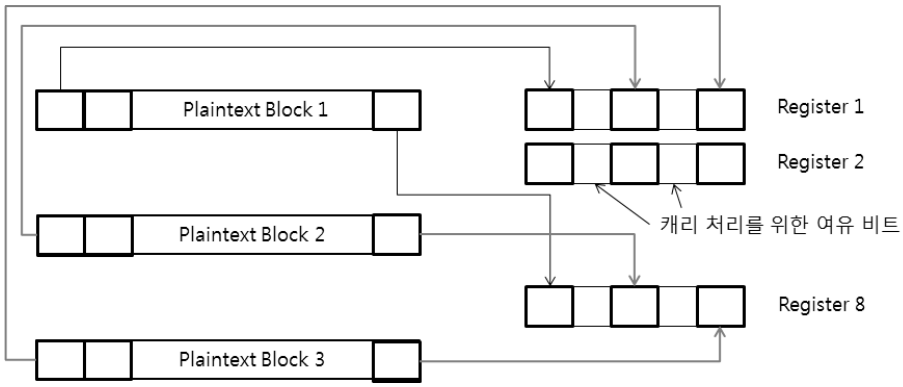
F_0, F_1 함수의 경우는 기존 구현에서 table lookup 방식으로 구현되어 아주 적은 비용이 소요된다. 하지만 비트 슬라이싱 기법에서 table lookup 방식을 사용하기 위해서는 데이터 형태를 바꿔야 하기 때문에 효율적이지 못하다. 이를 비트 슬라이싱 기법으로 효율적으로 구현하려면 논리회로 형태로 구현하면 된다. F_0, F_1 함수는 3번의 순환 이동 연산과 2번의 XOR 연산으로 이뤄져 있다. 앞에서 언급했듯이 순환 이동 연산은 비용이 들지 않기 때문에 F_0, F_1 함수는 16개의 XOR 연산만으로 구현이 가능하다. 따라서 실제 블록 하나 당 연산 횟수는 32비트 환경에서 0.5번, 64비트 환경에서 0.25번으로 table lookup 방식보다 빠르다.

이와 같이 비트 슬라이싱을 이용하면 대부분의 연산들이 기존 구현보다 빠른 속도를 갖기 때문에 이전보다 더욱 빠른 구현이 가능하다.

3.2 바이트 슬라이싱 구현

바이트 슬라이싱 기법은 여러 블록을 바이트 단위로 묶어서 바이트 단위로 병렬 연산을 하는 기법이다. 데이터의 형태가 바이트 단위일 뿐 기본적인 원리는 비트 슬라이싱과 유사하다. 각 블록의 첫 번째 바이트를 묶어서 첫 번째 데이터를 형성하고, 두 번째 바이트들을 묶어서 두 번째 데이터를 형성하고, 이와 같은 방식으로 n 번째 바이트까지 데이터를 형성한다. 이와 같이 여러 블록의 바이트들을 순서대로 묶어놓은 데이터를 입력으로 받아 바이트 단위의 연산을 한다. 바이트 슬라이싱 역시 레지스터의 크기가 크면 더 많은 블록을 동시에 연산할 수 있으므로 레지스터의 크기가 클수록 더 효율적인 구현이 가능해진다. 또한 바이트 슬라이싱은 비트 슬라이싱과 마찬가지로 F_0, F_1 함수 구현에서 연산량이 기존 구현보다 많아지지만 여러 개의 블록이 동시에 연산되므로 실제로는 더 빠른 성능을 보여준다.

본 논문에서는 HIGHT의 바이트 슬라이싱을 비트 슬라이싱과 마찬가지로 32비트 OS 환경과 64비트 OS 환경에서 구현하였다. 32비트 환경의 경우 하나



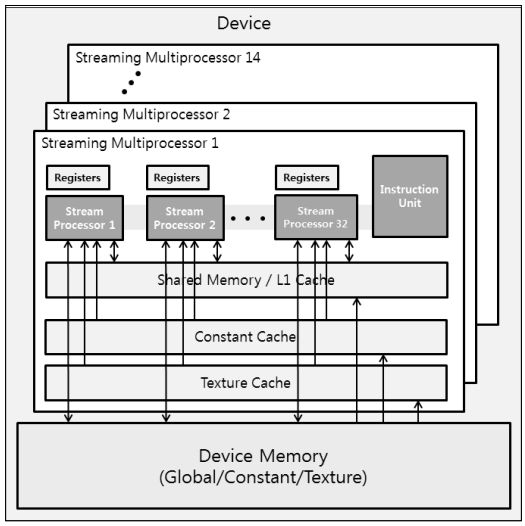
〔그림 5〕 32비트 OS 상에서의 HIGHT 바이트 슬라이싱 데이터 구조

의 레지스터에 네 개의 바이트를 담을 수 있으나, 실제로는 덧셈 연산에서 캐리가 발생하여 다른 블록의 바이트 값에 영향을 끼칠 수 있으므로 각 바이트 사이에 최소 1비트의 여유 공간이 주어져야 한다. 따라서 32비트 환경에서의 구현에서는 이 여유 공간을 주기 위해서 3바이트만 데이터를 위해 사용하고 나머지 1바이트를 각 바이트 사이에 4비트씩 나눠서 넣어주었다. 64비트 환경에서의 구현은 여유 공간을 위해 1바이트를 남겨두고 7바이트를 데이터 연산을 위해 사용한다. 그리고 남겨놓은 1바이트를 각 바이트 사이에 1비트씩 나눠서 넣어주었다.

XOR와 덧셈 연산은 32비트 환경과 64비트 환경 모두 같은 방식으로 연산한다. XOR는 32비트 환경에서는 3개의 블록이 동시에 연산이 되므로 블록 한 개 당 연산의 횟수는 약 0.34번이며, 64비트 환경에서는 7개의 블록이 동시에 연산되므로 블록 한 개 당 연산의 횟수는 약 0.14번이다. 덧셈의 경우 덧셈 연산 후 발생된 캐리 값을 초기화 해주기 위해 마스크 연산을 한다. 기존 구현도 덧셈과 마스크 연산으로 이뤄져 있기 때문에 연산량의 차이가 없으나, 바이트 슬라이싱 구현에서는 여러 블록이 동시에 연산되므로 더 빠르다. 3개의 블록이 동시에 연산되는 32비트 환경에서는 블록 당 연산 횟수가 약 0.67번이고, 7개 블록이 동시에 연산되는 64비트 환경에서는 블록 당 연산 횟수가 약 0.29번이다. 기존 구현에서 블록 당 2번의 연산을 하는 것과 비교하면 속도 개선 효과가 큰 것을 쉽게 확인할 수 있다.

순환 이동 연산은 쉬프트 연산 2번과 마스크 연산 2번으로 이루어진다. 예를 들어 왼쪽으로 3비트를 순환 이동하는 연산을 한다면 데이터를 왼쪽으로 3비트 쉬프트하고, 쉬프트로 인해 3비트만큼 왼쪽으로 이동한 데이터가 다른 블록의 데이터에 영향을 끼칠 수 있으므로 마스크 연산을 해준다. 이 연산 후 상위 5비트에 대한 데이터만 얻게 되므로, 하위 3비트의 값을 얻기 위해 오른쪽으로 5비트 쉬프트하고 마스크 연산을 해준다. 순환 이동 연산이 사용되는 F_0, F_1 함수는 기존 구현에서 table lookup 방식을 이용하여 구현되었다. 32비트 환경에서의 바이트 슬라이싱 구현 역시 순환 이동 연산을 통한 on-the-fly 계산보다 table lookup 방식을 이용하는 것이 더 효율적이다. table lookup을 하기 위해서는 여러 블록을 묶어놓았던 데이터를 각각 블록별로 쪼갰다가 합치는 연산이 필요한데, 이 데이터 변환을 위해 연산량이 늘어나기는 하지만 순환 이동 연산을 이용해서 직접 값을 계산해내는 것보다 연산량이 적다. 구체적으로, 순환 이동 연산을 하면 14번의 연산이 필요하지만 table lookup 방식으로 연산을 하면 데이터 변환을 위한 8번의 연산과 3번의 메모리 접근이면 된다. 하지만 64비트 환경에서는 table lookup 방식을 이용하기 위해서 7개의 데이터를 변환하여야 하기 때문에 순환 이동 연산을 이용할 때보다 연산의 횟수가 많아지게 된다. 따라서 64비트 바이트 슬라이싱 구현에서는 순환 이동 연산을 이용해 직접 F_0, F_1 함수의 값을 계산한다. 64비트 환경의 구현에서는 묶어놓은 7개의 블록 데이터들 사이에 1비트의 여유 공간이 있으므로 1비트 차이의 쉬프트 연산들을 묶어서 처리하면 마스크 연산의 횟수를 줄일 수 있으며, 결과적으로 64비트 환경에서 F_0과 F_1의 값을 얻기 위해 15번의 연산이 필요하다.

라운드 함수 단위로 살펴보면 기존 구현은 한 라운드 당 연산 횟수가 28번이고, 32비트 구현은 약 18.67번, 64비트 구현은 약 10.29번으로 기존 구현보다 더 적은 연산량을 보여준다. 이는 F_0, F_1의 값을 얻기 위



〔그림 6〕 GTX470 GPU의 구조

해 연산의 횟수가 많이 늘어났지만 덧셈과 XOR 연산에서 연산의 횟수가 많이 줄어들기 때문이다.

IV. GPU상의 HIGHT 고속화 구현

본 장에서는 GPU 환경에서의 HIGHT 고속화 구현에 대해 다룬다. KISA에서 제공해주는 소스코드와 앞 장에서 다뤘던 바이트 슬라이싱 구현을 NVIDIA의 CUDA 라이브러리[14]를 이용해 GPU 프로그램으로 포팅하는 방법으로 구현하였다. GPU는 레지스터의 크기가 32비트로 크지 않고, 개수도 제한적이어서 비트 슬라이싱 구현에 적합하지 않으므로 비트 슬라이싱 기법의 GPU 구현은 고려하지 않는다.

4.1 GPU의 구조

이 절에서는 본 논문에서 구현 및 실험에 사용한 NVIDIA의 GTX470 모델을 기준으로 GPU의 구조에 대해 살펴본다.

GTX470은 14개의 스트리밍 멀티프로세서(Streaming Multiprocessor, SM)를 갖고, 각 멀티프로세서에는 32개의 스트림 프로세서(Stream Processor, SP)가 있다. 각 스트리밍 멀티프로세서는 공유 메모리(shared memory)와 L1 캐시를 위한 64KB의 온 칩(on-chip) 메모리를 갖고, 이 메모리를 48KB 공유 메모리와 16KB L1 캐시 또는 16KB 공유 메모리와 48KB L1 캐시 등 필요에 따라 설정을 바꿔가며 사용할 수 있다. 공유 메모리는 스트리밍 멀티프로세서 내의 스트림 프로세서들 사이에서 공유된다. GTX470과 같은 페르미 구조(Fermi architecture)는 기존 구조와 다르게 전역 메모리에도 캐시 메모리를 가지고 있어 빠른 속도를 유지하면서 더 여유로운 메모리 사용이 가능하다. 각 스트리밍 멀티프로세서는 SIMT (Single Instruction Multiple Thread) 방식으로 동작하고, 하나의 멀티프로세서에서 최대 1536개의 쓰레드가 동시에 연산을 수행할 수 있다. GPU는 연산기(ALU)가 대부분을 차지하고 제어부분이 거의 없으므로 분기가 없고 연산이 많은 프로그램에서 좋은 성능을 얻을 수 있다.

4.2 CUDA 쓰레드 계층 구조 및 메모리 구조

CUDA의 쓰레드 계층 및 메모리 구조는 〔그림7〕과 같다. 호스트 PC에서 호출되는 GPU의 서브 프로그램을 커널(kernel)이라 부르며, 커널은 블록들로 구성된 그리드이다. 블록들은 여러 개의 쓰레드로 구성되며, 하나의 블록은 한 스트리밍 멀티프로세서에 할당된다. 공유 메모리와 레지스터 자원은 블록 내에서 공동으로 활용하며, 각 스트리밍 멀티프로세서는 자원의 여유가 있으면 여러 개의 블록을 실행한다.

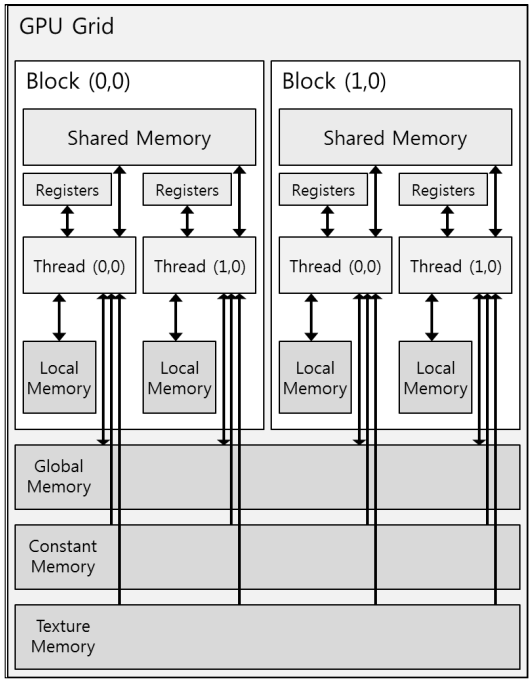
전역 메모리, 콘스탄트 메모리, 텍스쳐 메모리는 모든 블록의 쓰레드들이 접근할 수 있으며, 공유 메모리는 같은 블록 내에 존재하는 쓰레드들만 접근할 수 있다. 레지스터는 쓰레드마다 필요한 크기만큼 할당되어 각 쓰레드는 자신에게 할당된 레지스터만 접근할 수 있다.

4.3 GPU상의 HIGHT 구현

GPU상의 HIGHT 구현은 KISA에서 제공해주는 소스코드와 앞 장에서 다뤘던 바이트 슬라이싱 코드를 CUDA 라이브러리를 이용하여 GPU 프로그램으로 포팅하는 방식으로 구현하였다.

CUDA는 C언어의 확장으로 설계되었기 때문에, GPU에서의 구현 역시 앞장에서 다뤘던 CPU 구현과 주요 구현 부분이 모두 같다. 그러므로 이절에서는 메모리 사용 등 같이 GPU 프로그램의 성능에 큰 영향을 미치는 요소들 위주로 설명한다.

GPU 프로그램을 구현하려면 먼저 입력으로 사용되는 데이터들을 어떤 메모리에 넣어 사용할 것인지 결정해야 한다. 온 칩 메모리인지 오프 칩 메모리인



〔그림 7〕 CUDA 쓰레드 계층 및 메모리 구조

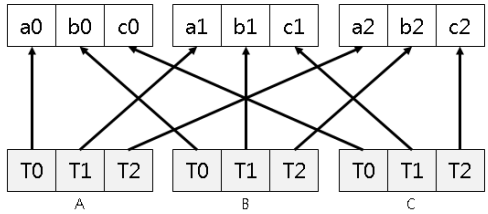
지, 캐시 사이즈가 얼마나 되는지 등에 따라서 메모리별로 성능의 차이가 있으므로 데이터의 크기와 활용도를 따져 적당한 메모리를 사용하는 것이 중요하다.

입력으로 들어오는 평문과 최종 출력으로 나오게 되는 암호문은 처음과 마지막 한 번씩만 메모리에 접근하게 되므로 전역 메모리를 이용한다. 전역 메모리에서 데이터를 읽어올 때 캐시 히트(cache hit) 효과를 최대화하기 위하여 통합 접근(coalesced access) 기법을 적용하여 구현하였다.

통합 접근 기법을 적용하지 않으면 다음 코드와 같이 구현되며, 〔그림 8〕과 같이 접근하게 된다.

```
int p[] = {a0, b0, c0, a1, b1,
                       c1, a2, b2, c2};
__global__ void kernel( int *p){
  int *ptr = p + threadIdx.x*3
             + blockIdx.x*blockdim.x*3;
  int x = ptr[0];   // A
  int y = ptr[1];   // B
  int z = ptr[2];   // C
}
```

한 개의 쓰레드의 입장에서는 순차적으로 데이터를 처리하지만 한 번에 여러 쓰레드들이 동시에 실행되므로 한 쓰레드가 처리해야 하는 데이터 크기만큼 떨어져있는 데이터에 접근하게 되어 캐시 히트 효과를 볼 수 없게 된다.



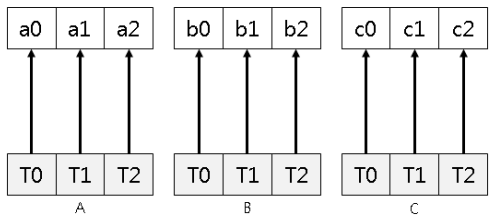
〔그림 8〕 Uncoalesced access

통합 접근 기법을 적용한 코드는 다음과 같으며, 〔그림 9〕와 같이 접근하게 된다.

```
int p[] = {a0, a1, a2, b0, b1,
                       b2, c0, c1, c2};
__global__ void kernel( int *p){
  int *ptr = p + threadIdx.x
             + blockIdx.x*blockdim.x*3;
  int x = ptr[0];   // A
  int y = ptr[1*blockdim.x];   // B
  int z = ptr[2*blockdim.x];   // C
}
```

통합 접근 기법을 적용하여 구현하면 〔그림 9〕와 같이 모든 쓰레드들이 인접해있는 데이터를 읽게 되어 캐시 히트 효과를 최대화 할 수 있다.

암호화 연산에 필요한 F_0, F_1 함수의 테이블과 화이트닝키, 서브키 등의 키 데이터는 암호화 연산중에 자주 접근하게 되므로 빠른 접근이 가능한 온 칩 메모리인 공유 메모리를 사용한다. 공유 메모리를 사용하기 위해서는 커널 실행 후 오프 칩 메모리에 저장된 데이터를 공유메모리로 복사하는 연산이 필요하다. 이 연산이 자칫 오버헤드로 작용할 수도 있지만 공유 메모리에서 사용되는 데이터는 주로 같은 블록 내에 속한 모든 쓰레드들이 함께 공유하므로 복사할 때 각 쓰



〔그림 9〕 Coalesced access

레드 별로 복사할 데이터를 나눠 할당해 주면 쓰레드 당 한 번 또는 두 번 정도의 데이터 복사만 하면 된다. 평문 데이터를 읽어와 연산을 할 때 빠른 연산을 하기 위해 레지스터를 활용하는데, HIGHT는 64비트의 데이터가 바이트 단위로 연산되므로 이를 위해 8개의 레지스터를 사용한다.

블록 당 쓰레드의 개수 또한 성능에 영향을 미치는 중요한 변수이다. 보통은 GPU의 자원이 제한적이기 때문에 쓰레드 개수에 따라 달라지는 자원을 비교하여 최적의 조합을 찾아내는 방법을 사용하여 결정한다. GTX470의 경우는 비교적 최신모델로 자원이 여유롭기 때문에 HIGHT 구현 시 어떤 조합을 사용하더라도 필요한 자원이 모두 지원되는 것을 확인할 수 있다. 다만, 한 개의 멀티프로세서가 최대 1536개의 쓰레드를 사용할 수 있으므로, 쓰레드의 개수를 192, 256, 512, 768 등의 크기로 선언하여 구현해야 모든 쓰레드를 사용할 수 있어 최적의 성능을 보여준다. 한편 F_0, F_1 함수를 공유 메모리에 복사할 때 오버헤드를 최소로 하기 위해 256개 이상의 쓰레드를 선언하여 사용하는 것이 좋다. 256개 이상의 쓰레드를 사용하면 각 테이블을 복사하기 위해 한 쓰레드가 한 개의 데이터만 복사하면 된다. 256보다 작은 192를 제외하고 최적의 성능을 발휘할 수 있는 256, 512, 768의 값을 실험적으로 적용하여 쓰레드의 개수를 결정하였다. 실험 결과 기존 구현과 바이트 슬라이싱 기법을 적용한 구현 모두 768의 쓰레드 개수를 선언했을 때 더 좋은 성능을 보여주어 이 결과를 이용하였다.

V. 실험 결과

실험은 2.67GHz의 클럭을 지원하는 Intel core i7 920 CPU와 607MHz 클럭 및 1280MB 메모리를 지원하는 NVIDIA GeForce GTX 470 GPU 환경에서 수행하였다. 수행한 결과는 키 스케줄링을 포함하지 않는 순수 암호화 연산에 대한 속도이다.

CPU 환경에서의 실험 결과는 〔표 1〕과 같다. 32비트 환경과 64비트 환경에서 모두 기존 구현보다 비트 슬라이싱, 바이트 슬라이싱 구현이 더 좋은 성능을 보여주는 것을 볼 수 있다. 또한 이 결과를 통해 레지스터의 크기가 클수록 더 좋은 성능을 얻을 수 있음을 확인할 수 있으며, 특히 64비트 환경에서는 바이트 슬라이싱보다 비트 슬라이싱이 월등한 성능을 보임을 확인할 수 있다. 결과적으로 비트 슬라이싱을 이용한 구현은 KISA의 소스코드와 비교하여 32비트 환경에서는 약 1.7배, 64비트 환경에서는 약 2.4배의 성능을 보여준다.

〔표 1〕 CPU 환경에서의 실험 결과

구현 방법	32비트 운영체제	64비트 운영체제
기존 구현	0.38Gbps	0.62Gbps
비트 슬라이싱	0.66Gbps	1.48Gbps
바이트 슬라이싱	0.59Gbps	0.98Gbps

〔표 2〕 GPU 환경에서의 실험 결과

구현 방법	비통합 접근	통합 접근
KISA 소스코드 포팅	36Gbps	38Gbps
바이트 슬라이싱	40Gbps	46Gbps

GPU 환경에서의 실험 결과는 〔표 2〕와 같다. 통합 접근 방식을 적용해 수행한 결과가 비통합 접근 방식으로 수행한 결과보다 더 향상된 결과를 얻을 수 있음을 확인할 수 있으며, 바이트 슬라이싱 기법을 적용한 구현이 KISA 소스코드를 포팅한 방법보다 더 빠른 결과를 보여주는 것을 확인할 수 있다. 바이트 슬라이싱 기법과 통합 접근 기법을 함께 적용한 구현이 가장 빠른 결과를 보여주며, 통합 접근 기법을 적용하여 KISA 소스코드를 포팅한 구현보다 약 1.2배 더 빠른 결과를 보여주는 것을 볼 수 있다. 한편, 〔표 1〕과 〔표 2〕의 결과를 비교해보면 GPU의 성능이 CPU에 비해 약 31배 가량으로 월등히 높은 것을 볼 수 있다.

VI. 결 론

본 논문에서는 비트 슬라이싱 및 바이트 슬라이싱 기법을 적용하여 CPU와 GPU 환경에서 각각 HIGHT 알고리즘을 최적화 구현하였다. 비트 슬라이싱 및 바이트 슬라이싱 기법을 CPU에서 구현한 결과 기존 구현보다 최대 2.4배의 성능을 얻을 수 있었다. CPU 구현은 32비트 OS 환경과 64비트 OS 환경으로 나눠서 구현 및 실험하였으며, 이들에 대한 결과 비교를 통해 레지스터의 크기가 클수록 더 좋은 성능을 보여주는 것 또한 확인할 수 있었다. GPU 구현의 경우 레지스터 크기가 32비트로 비교적 크지 않고 레지스터의 개수 또한 제한적이므로 비트 슬라이싱 기법의 적용이 적합하지 않으며, 비교적 적은 양의 레지스터만으로 구현이 가능한 바이트 슬라이싱을 이용할 경우 KISA 소스코드를 단순히 포팅한 것에 비해 약 1.2배 빠름을 확인할 수 있었다.

참고문헌

〔1〕 한국인터넷진흥원, "HIGHT 블록암호 알고리즘 사양 및 세부 명세서," http://seed.kisa.or.kr/kor/hight/hightInfo.jsp

〔2〕 D. Hong, J. Sung, S, Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim and S. Chee, "HIGHT: A new block cipher suitable for low-resource device," *CHES2006*, LNCS 4249, pp. 46-59, 2006.

〔3〕 International Organization for Standardization, "ISO/IEC 18033-3:2010," ISO/IEC 18033-3, 2010.

〔4〕 National Institute of Standards and Technology, "Data Encryption Standard (DES)," FIPS PUB 46-2, Dec 1993.

〔5〕 National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," FIPS PUB 197, Nov 2001.

〔6〕 한국인터넷진흥원, "SEED 블록암호 알고리즘," http://seed.kisa.or.kr/kor/seed/seedInfo.jsp

〔7〕 한국인터넷진흥원, "ARIA 블록암호 알고리즘," http://seed.kisa.or.kr/kor/aria/aria.jsp

〔8〕 S.A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," ICSPC2007, pp. 65-68, Nov. 2007.

〔9〕 D.A. Osvik, J.W. Bos, D. Stefan and D. Canright, "Fast software AES encryption," FSE2010. LNCS 6147, pp. 75-93, 2010.

〔10〕 염용진, 조용국, "GPU용 연산 라이브러리 CUDA를 이용한 블록암호 고속 구현," 정보보호학회논문지, 18(3), pp. 23-32, 2008년 6월.

〔11〕 임지혁, 강정민, 조수민, 김현오, 김동규, "GPU용 라이브러리 CUDA를 이용한 SEED 알고리즘의 고속화 구현," 한국정보과학회 학술발표논문집, 37(2B), pp. 417-421, 2010년.

〔12〕 E. Biham, "A fast new DES implementation in software," FSE 1997, LNCS 1267, pp. 260-272, 1997.

〔13〕 E. Kasper and P.Schwabe, "Fast and timing-attack resistant AES-GCM," CHES2009. LNCS 5747, pp. 1-17, 2009.

〔14〕 NVIDIA, "NVIDIA CUDA C programming guide," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

〈著者紹介〉

백 은 태 (Eun-Tae Baek) 학생회원
2010년 2월: 인하대학교 정보공학계열 학사
2010년 9월~현재: 인하대학교 컴퓨터정보공학 석사과정
〈관심분야〉 정보보호, 암호학, 최적화 구현 등.

이 문 규 (Mun-Kyu Lee) 종신회원
1996년 2월: 서울대학교 컴퓨터공학과 학사
1998년 2월: 서울대학교 컴퓨터공학과 석사
2003년 8월: 서울대학교 컴퓨터공학과 박사
2003년 8월~2005년 2월: 한국전자통신연구원 선임연구원
2005년 3월~현재: 인하대학교 컴퓨터정보공학부 부교수
〈관심분야〉 정보보호, 암호학, 컴퓨터이론 등.