情報保護學會論文誌
第18卷 第3號, 2008. 6

GPU용 연산 라이브러리 CUDA를 이용한 블록암호 고속 구현

염 용 진[1†], 조 용 국[2]
[1]ETRI 부설 연구소, [2]한양대학교

High-Speed Implementations of Block Ciphers on Graphics Processing Units Using CUDA Library

Yongjin Yeom[1†], Yongkuk Cho[2]
[1]The Attached Institute of ETRI, [2]Hanyang University

요 약

그래픽 프로세서(GPU)의 연산 능력은 이미 CPU를 능가하고 있으며, 그 격차는 점점 벌어지고 있다. 따라서, 범용 계산에 그래픽 프로세서를 활용하는 GPGPU 연구가 활발히 전개되고 있으며, 병렬 처리가 필요한 분야에서 특히 두드러진 성과를 보이고 있다. GPU를 이용한 암호 알고리즘의 구현은 2005년 Cook 등에 의하여 처음 시도되었으며, OpenGL, DirectX 등의 라이브러리를 이용하여 개선된 결과들이 속속 발표되고 있다.

본 논문에서는 2007년 발표된 NVIDIA의 CUDA 라이브러리를 이용한 블록암호 구현 기법과 그 결과를 소개하고자 한다. 또한, 소프트웨어로 구현된 블록암호 소스를 GPU 프로그램으로 이식하는 일반적인 방법을 제공하고자 한다. 8800GTX GPU에서 블록암호 AES, ARIA, DES를 구현했으며, 속도는 각각 4.5Gbps, 7.0Gbps, 2.8Gbps로 CPU보다 고속 구현이 가능하였다.

ABSTRACT

The computing power of graphics processing units(GPU) has already surpassed that of CPU and the gap between their powers is getting wider. Thus, research on GPGPU which applies GPU to general purpose becomes popular and shows great success especially in the field of parallel data processing. Since the implementation of cryptographic algorithm using GPU was started by Cook et al. in 2005, improved results using graphic libraries such as OpenGL and DirectX have been published.

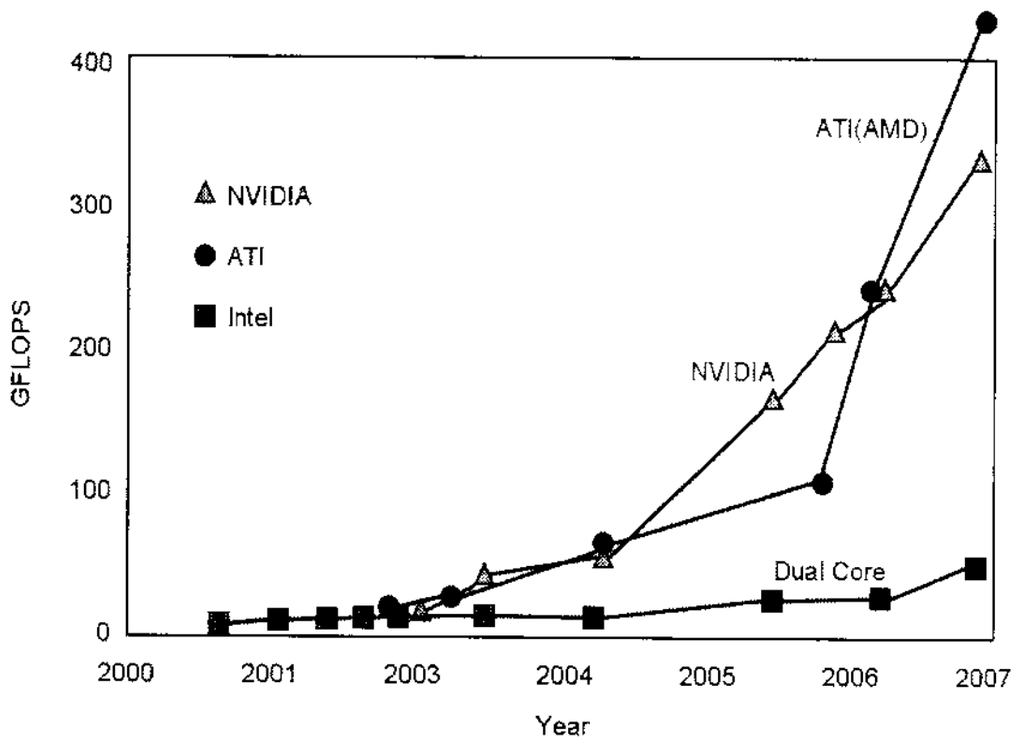
In this paper, we present skills and results of implementing block ciphers using CUDA library announced by NVIDIA in 2007. Also, we discuss a general method converting source codes of block ciphers on CPU to those on GPU. On NVIDIA 8800GTX GPU, the resulting speeds of block cipher AES, ARIA, and DES are 4.5Gbps, 7.0Gbps, and 2.8Gbps, respectively which are faster than the those on CPU.

Keywords : Block Cipher, CryptoGraphics, Graphics Processing Unit, GPGPU, CUDA, AES, ARIA, DES

접수일 : 2007년 12월 16일; 수정일 : 2008년 2월 28일;
채택일 : 2008년 3월 25일
† 주저자, yjyeom@ensec.re.kr

Ⅰ. 서 론

그래픽 카드는 그래픽 프로세서(GPU, Graphics Processing Unit)와 메모리를 가지고 있으며, 컴퓨터에서 요구되는 그래픽 관련 부동 소수점 연산들을 주로 담당한다. 1994년 3dfx사가 3차원 그래픽용 부두(Voodoo)를 발표한 이래, 멀티미디어와 게임의 인기에 힘입어 그래픽 프로세서는 비약적인 발전을 거듭해 오고 있다. [그림 1]은 그래픽 프로세서와 인텔 CPU의 부동소수점 연산 능력을 비교한 것으로 이미 GPU의 성능은 CPU를 압도하고 있음을 보여준다.



〔그림 1〕 CPU와 GPU의 연산 능력 비교〔1〕

하지만 그래픽을 처리하지 않는 동안은 GPU가 가진 연산 능력의 대부분이 활용되지 못하고 만다. 이러한 유휴 자원인 GPU의 컴퓨팅 파워를 범용 계산에 활용하는 연구인 GPGPU (General Purpose computations on GPU)[2]가 수학 및 과학 분야에서 급속히 확산되고 있다. 특히, GPU의 구조는 작은 프로세서의 배열로 되어 있기 때문에 병렬 연산에서 탁월한 성능을 보여주고 있다. 행렬 연산, 데이터 정렬(sort) 등 기본 연산에서 이미 CPU보다 고속 처리가 가능하며, 다체 문제(N-body problem), 바이오 정보처리 등의 응용에서도 효과적임을 입증하는 논문이 다수 발표되었다[3,4]. 또한, 최근에는 여러 개의 그래픽 카드를 내장하여 병렬 연산을 더욱 가속화하는 동작 방식도 지원되어, 다수의 GPU를 탑재한 데스크탑 수퍼컴퓨팅 시대의 도래가 예상된다. [표 1]은 경제적인 측면을 고려할 때 연산 당 소요되는 비용에서도 압도적으로 GPU가 우세함을 보여주고 있어, 앞으로 GPU의 연산 능력을 이용하는 것은 크게 일반화될 전망이다.

〔표 1〕 최신 CPU와 GPU의 비교〔1,5〕

프로세서	Intel Core2 Quad (QX6850)	NVIDIA Geforce 8800GTX
연산능력	96 GFLOPS	330 GFLOPS
메모리 대역폭	21GB/s	55.2GB/s
가격	$1100(chip)	$550(board)
발전속도 (매년)	x 1.4	x 1.7(pixel) x 2.3 (vertex)

GPU의 우수한 연산 잠재력에도 불구하고, GPU의 활용은 그동안 제한적인 범위에서만 가능하였다. 왜냐하면, 그래픽을 위해 최적화된 설계로 범용 연산을 위한 명령어가 부족하고, GPU 프로그래밍은 그래픽 라이브러리를 통해서만 가능했기 때문이다. 그러나 최근 수년간 있어온 다음과 같은 변화들은 GPU의 활용을 촉진하고 있다.

① CPU/GPU 발전속도의 불균형 : CPU의 발전 방향이 연산능력의 증대에서 멀티 코어의 탑재로 전환되고 있어 CPU의 속도 증대는 다소 주춤한 상태이다. 반면 GPU의 계산능력은 매년 약 2배 정도 증가하고 있으며 GFLOPS로만 비교한다면 CPU의 처리능력을 압도하고 있다.

② 인터페이스의 속도개선 : GPU의 계산 능력이 탁월해도 데이터를 전송하는 인터페이스가 느리다면 활용 범위는 비교적 자료 전송량이 적은 분야에 한정된다. 하지만, 그래픽 카드의 인터페이스가 AGP에서 PCI express로 바뀌면서 대역폭(bandwidth)이 8GB/s까지 증가하여 병목현상이 거의 제거되었다. 이는 블록암호를 이용한 암호화와 같이 다량의 데이터를 고속으로 처리하는 데도 GPU가 적극 개입할 수 있음을 의미한다.

③ 범용 라이브러리의 발표 : 한동안 GPU의 연산을 활용하기 위해서는 그래픽에 대한 어느 정도의 지식이 불가피 하였다. OpenGL[6] 그래픽 라이브러리를 이용하는 경우, 프레임 버퍼로부터 자료를 받아 pixel에 대한 연산을 수행하거나, 텍스쳐(texture), 쉐이딩(shading) 작업을 활용해야만 했다. 이를 개선하기 위해 2007년 NVIDIA는

CUDA 라이브러리를 발표함으로써 GPU를 병렬 프로세서로 간주할 수 있는 범용 프로그래밍 환경을 제공하기 시작하였다. 최근 발표되는 GPGPU 관련 연구 결과의 다수가 CUDA 라이브러리를 활용하고 있다.

④ 범용 연산의 지원 : 초기의 GPU는 부동소수점 연산을 위한 가속기로서, 정수연산도 제공하지 않아 잠재적인 계산 능력을 활용하기에 어려움이 많았다. 최신 GPU는 정수연산, 비트연산 등 암호 알고리즘의 구현에 필요한 대부분의 연산을 내장하고 있을 뿐만 아니라, 삼각함수 등과 같은 초월함수의 일부도 자체적으로 처리하고 있다.

이러한 컴퓨팅 환경의 변화에 따라 암호 구현 분야에서도 실용적인 결과가 도출되기 시작하였다. Cook 등[7,8]에 의하여 최초로 GPU를 이용한 AES의 구현이 발표되었을 당시에는 그 속도가 CPU의 1/100에 불과하여 가능성을 제시하는 것에만 의의가 있었다. CHES 2007[9]에서는 OpenGL의 텍스쳐 프로그래밍을 이용하면 AES의 속도가 CPU를 능가할 수 있음이 발표되었고, Yamanouchi[10]는 NVIDIA의 GeForce G80에 적용되는 OpenGL extension 기능을 이용하여 3Gbps 급의 구현 결과를 제시하였다. AMD(구 ATI)의 Yang 등[11]은 자사의 GPU인 HD 2900 XT에서 AES 키 탐색을 비트 슬라이스 기법으로 구현하는 경우 30 Gbps에 도달할 수 있음을 ASIACRYPT 2007에서 주장하였다. 이 결과는 DirectX와 ATI CTM 및 GPU 어셈블리를 이용한 최적화를 통하여 얻은 것이다.

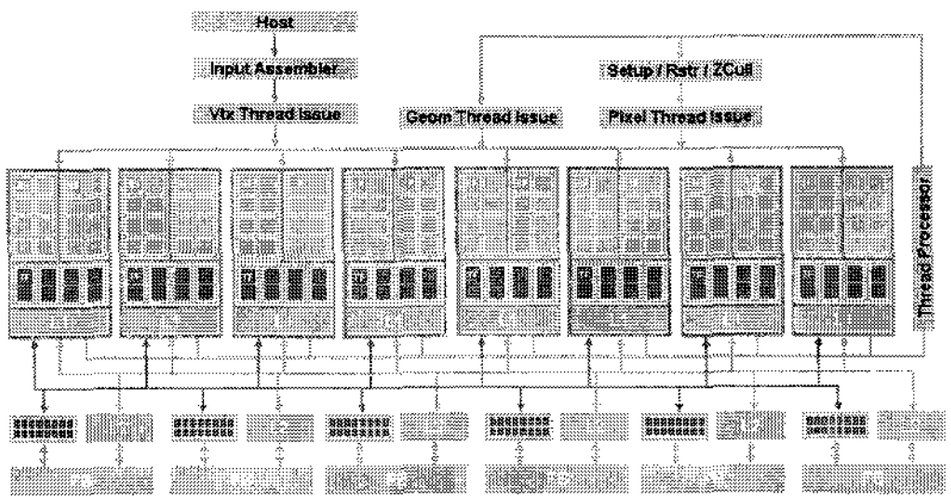
위의 결과들은 OpenGL, DirertX 등의 그래픽 라이브러리와 각 GPU 제조사에서 제공하는 추가적이 확장 라이브러리, 어셈블리어 등을 이용한 것이다. 이를 위해서는 GPU상에서의 그래픽 처리과정에 대한 이해가 필요하며, 각 제조사별 최적화는 좋은 성능을 제공하지만 호환성을 떨어뜨리게 된다.

본 논문에서는 2007년 NVIDIA에서 발표한 CUDA (Compute Unified Device Architecture)[12] 라이브러리를 이용하여 블록암호를 구현하는 방법과 AES, ARIA, DES의 구현 결과를 제시하고자 한다. 또한, 블록암호의 종류에 무관하게 기존의 CPU용 프로그램을 GPU 프로그램으로 변환하는 틀을 제공하여 다른 블록암호에도 쉽게 활용할 수 있도록 하였다.

Ⅱ. GPU의 구조와 CUDA 라이브러리

2.1 GPU의 구조

GPU는 호스트 컴퓨터로부터 받은 데이터를 프레임 버퍼에 넣는 과정에서 필요한 일련의 32비트 부동소수점 연산을 빠르게 수행하기 위한 병렬 구조를 가지고 있다. 여기서는 CUDA 라이브러리를 수행하기 위한 기본 환경인 NVIDIA의 G80 시리즈를 중심으로 살펴본다.



〔그림 2〕 NVIDIA의 G80 그래픽 카드의 구조

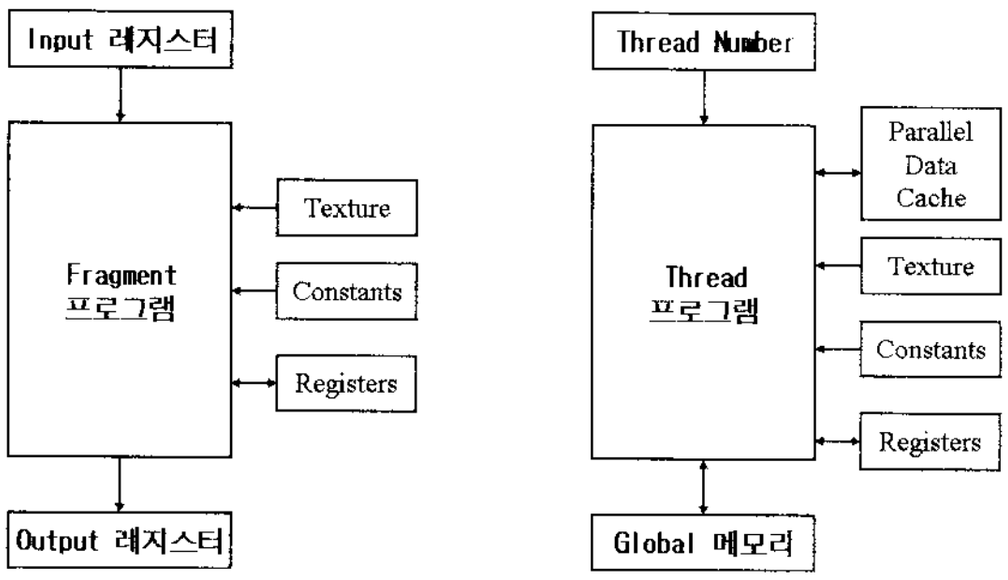
NVIDIA의 GeForce 8800GTX는 [그림 2]에서와 같이 프레임 버퍼에 스트리밍 멀티 프로세서(Streaming Multiprocessor, MP)가 연결되어 있고, 각 멀티 프로세서에는 8개의 스트림 프로세서(Stream Processor, SP)가 있다. 각 멀티 프로세서의 16KB 공유 메모리(shared memory)를 스트림 프로세서들이 공유하며, L1, L2 캐시 메모리도 가지고 있다. 4단계의 파이프 라인 구조를 모두 활용하면 하나의 멀티 프로세서에서 최대 32개의 쓰레드가 동시에 연산을 수행할 수 있다. GeForce 8x00 시리즈는 다양한 성능의 GPU를 제공하고 있으나, 범용 계산에 가장 널리 사용되고 있는 것은 GeForce 8800GTX로 768MB의 그래픽 메모리를 가진다. 각 스트리밍 멀티 프로세서는 SIMD (Single Instruction Multiple Data) 방식으로 동작하며 20 GFLOPS 이상의 성능을 보여준다. CPU와 비교하면 GPU는 제어부분이 거의 없고 연산기(ALU)가 대부분을 차지하는 구조로서, 분기가 없으며 연산이 많은 프로그램의 실행시 탁월한 성능을 보여준다[12].

2.2 CUDA 라이브러리

앞절에서 기술한 바와 같이 GPU 프로그래밍은 그래

픽 분야의 비전문가에게는 많은 장벽이 있어 왔다. 초기의 GPGPU 방식에서는 텍스쳐는 배열, 쉐이딩은 커널에 대응시키는 프로그래밍을 해왔다[13]. 보다 쉬운 활용을 위해 고급 언어로 GPU의 병렬처리를 지원하는 방법으로 Cg(NVIDIA), HLSL(Microsoft), Brook (Stanford 대학), GLSL(OpenGL) 등 다양한 라이브러리가 발표되었다. 최근에는 NVIDIA의 CUDA와 AMD의 CTM이 발표되어 다른 도구를 압도하고 있다. 특히, CUDA의 경우 Matlab의 성능 향상을 위한 플러그 인(plug-in), 행렬연산 패키지인 CUBLAS, 정렬(sort)기법 등을 포함하는 병렬 기본연산 패키지 CUDPP 등이 속속 발표되고 있다.

CUDA는 NVIDIA가 2007년 발표한 범용 GPU 프로그래밍 라이브러리이다. 리눅스와 윈도우 환경에서 기존 C 컴파일러와 연동되는 방식으로 동작하며 아직까지 관련 문서도 미흡하고, 동작이 불안정하며, 디버깅 환경이 열악하지만, NVIDIA GPU의 성능을 최대한 활용할 수 있는 도구로 알려져 있다. CUDA 라이브러리를 이용하면, [그림 3]과 같이 기존의 fragment 프로그램을 멀티 쓰레드(thread) 기반의 SIMD 프로그래밍으로 전환할 수 있다.



〔그림 3〕 Fragment 프로그래밍과 CUDA 프로그래밍의 비교

CUDA는 GPU의 메모리를 온칩(on-chip)과 오프칩(off-chip)으로 나누어 관리한다. 메모리의 관리는 프로그램의 실행 방식을 결정하는 그리드(grid), 블록(block), 쓰레드(thread)의 구성에 크게 영향을 미친다. 메모리의 종류와 특징은 [표 2]과 같다. 메모리의 전송 지연을 줄이기 위해 가급적 온칩 메모리만을 사용하도록 하는 것이 실행 속도 향상에 도움이 되며, 적은 양의 온칩 메모리를 효율적으로 관리하는 것이 관건이다. 이는 메모리는 풍부하고 연산속도가 상대적으로 아쉬운 PC에 경우와 정반대이다.

CUDA 프로그램은 모든 쓰레드에 동일한 프로그램이 할당되며, 실행시 쓰레드의 고유번호(또는 좌표)가 파라미터로 사용되어 수행하는 내용이 달라지는 방식으로 작성된다. 이는 전형적인 SIMD 프로그램에 해당된다. 실제로 메모리의 계층구조를 활용하여 그리드와 쓰레드로 프로그램을 구성하는 것이 CUDA 프로그래밍에서 가장 핵심적인 부분에 해당한다.

〔표 2〕 CUDA에서 사용하는 GPU 메모리의 종류와 특징

메모리	위치	속도	특징
register	on chip	4 cycles	각 MP에 8,192개의 32비트 레지스터가 있음
shared		4 cycles	각 MP에 16KB의 공유 메모리가 있으며, 충돌지연이 없으면 레지스터와 같은 속도임
global /local device	off chip	high latency 400~600 cycles (80GB/s)	그래픽 카드내의 device 메모리(수백 MB)임 로컬 메모리는 글로벌 메모리의 일부를 쓰레드별 고유영역으로 할당함을 의미함
texture		high latency (cached)	읽기전용의 특별한 메모리
constant		low latency (cached)	64KB로 할당된 읽기전용 메모리
host	PC	최대 4GB/s	PC의 메인 메모리임

호스트 PC에서 호출되는 GPU 서브 프로그램을 커널(kernel)이라 부른다. 커널은 블록들로 구성된 그리드이며 블록은 다시 여러 개의 쓰레드로 구성된다. 커널을 직접 쓰레드의 모임으로 정의하지 않고, 중간에 블록의 개념을 도입하는 것은 여러 개의 스트리밍 멀티 프로세서를 가지는 GPU의 하드웨어 구조 때문이다. 하나의 블록은 한 멀티 프로세서에 할당되어 공유 메모리와 레지스터를 공동으로 활용하게 되어있으며, 멀티 프로세서는 자원의 여유가 있으면, 여러 개의 블록을 실행한다. 블록내의 쓰레드는 서로의 실행에 영향을 주지만, 다른 블록에 있는 쓰레드는 완전히 독립적이다. 즉, 데이터의 상호참조도 일어나지 않으며, 실행 순서도 특별히 정해지지 않는다.

Ⅲ. CUDA를 이용한 블록암호 구현

3.1 CUDA 프로그램의 기본 구조

블록암호를 구현하기 위하여 CUDA를 이용한 프로그램의 제어 흐름과 기본 틀에 대하여 알아보자. CUDA는 C언어의 확장으로 설계되었으며, GPU에서 실행되도록 작성된 커널은 그래픽 카드가 설치된 호스트 PC의 호출에 의해서 시작된다.

프로그램의 빌드 과정을 요약하면 다음과 같다. CUDA로 작성된 소스 코드(*.cu 파일)는 CUDA 컴파일러를 거쳐 기존 C 컴파일러에 전달된다. MS의 visual studio를 사용하거나 gcc를 쓰는 경우 CUDA 컴파일은 C언어의 전처리기(preprocessor)와 유사한 역할을 한다. 이 과정에서 GPU에서 실행될 함수들에 대한 GPU용 오브젝트 코드가 생성된다.

```
/* 호스트 PC에서 실행되는 소스 코드 */
// 커널 함수(항상 __global__ void로 선언)
__global__ void kernel_ftn(parameters)
{ (GPU용 코드) };

//kernel 구성을 위한 변수 선언
dim3 dimGrid(100,100);
dim3 dimBlock(10,10);

//GPU에 커널 함수의 실행을 요청
kernel_function
<<<dimGrid, dimBlock>>>(parameters);
```

GPU에서 실행될 커널 함수가 준비되면, 커널을 호출하기 전 그리드를 구성해야 한다. 위의 예는 100(10×10)개의 쓰레드로 구성된 블록을 10,000(100×100)개 가지는 그리드 구성을 보여준다. 이 경우 총 쓰레드의 개수는 1,000,000개가 실행되며, 한 블록에 들어있는 100개의 쓰레드는 서로 데이터를 공유하며 병렬 처리를 수행할 수 있지만, 블록들 사이의 실행은 완전히 독립적이다.

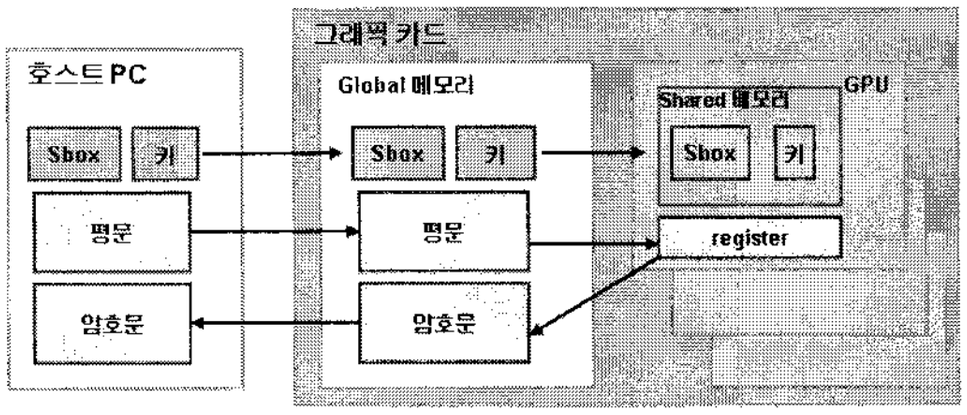
GPU로 프로그램에 필요한 데이터를 전달하는 방법은 커널 함수의 파라미터를 이용하는 방법과 글로벌(global) 메모리의 포인터를 이용하는 방법이 있다. 대량의 데이터는 메인 메모리에서 그래픽 카드의 메모리로 복사한 후 그 포인터를 파라미터로 전달한다. 커널 함수의 리턴값은 항상 void로 선언되어야 하기 때문에 실행 결과는 글로벌 메모리에 넣고, 호스트 PC에서 포인터로 읽어온다.

```
/* 호스트 PC에서 실행되는 소스 코드 */
int *host_ptr; //호스트 PC 메모리용 포인터
int *dev_ptr; //그래픽 카드 메모리용 포인터

//호스트 PC의 메모리 확보
host_ptr = (int *)malloc(size);
//그래픽 카드의 메모리 확보
cudaMalloc( (void**) &dev_ptr, size);

//데이터 복사(호스트 PC -> 그래픽 카드)
cudaMemcpy(dev_ptr, host_ptr,
            size, cudaMemcpyHostToDevice);
```

3.2 CUDA를 이용한 블록암호 구현

대부분의 블록암호가 공통적으로 Sbox와 라운드 키를 사용하고 있으므로, 호스트 PC에서 실행되는 부분은 유사한 구조를 가진다. 블록암호의 종류에 따라 Sbox의 개수와 데이터 타입이 달라지지만, 아래의 소스를 크게 벗어나지 않는다. 실제로 아래의 틀을 이용하여 블록암호 AES, ARIA, DES를 구현하였으며, 그 결과는 다음 절에서 다루도록 한다.

커널의 구성에 따라 암호화 속도가 큰 차이를 보일 수 있다. 시간 소모가 가장 많은 부분이 데이터의 전송이므로 가급적 많은 쓰레드를 사용하여 지연시간(latency time)동안 다른 쓰레드가 실행될 수 있도록 하는 것이 일반적으로 좋은 결과를 준다. 하지만 쓰레드의 실행 코드 크기에 따라 블록에 넣을 수 있는 개수가 달라지며, 최적화를 위해서는 많은 실험이 필요하다. 최적화에 대해서는 다음 절에서 다루도록 한다.



〔그림 4〕 호스트 PC와 그래픽 카드, GPU 사이의 데이터 전송

```
/* 호스트 PC에서 실행되는 소스 코드 */

// Sbox 전달(32비트 Sbox인 경우)
u32 hSbox[256]; //u32는 unsigned int임
/* Sbox 내용 설정 */
u32 *dSbox; //그래픽 카드용 포인터 선언
// 메모리 확보
cudaMalloc((void**) &dSbox, sizeof(u32)*256));
// Sbox 복사(host PC -> 그래픽 카드)
cudaMemcpy(dSbox, hSbox, 4*256,
                    cudaMemcpyHostToDevice);

// 평문, 라운드 키 전달
Byte hPT[16*numBlock];//블록크기*블록개수
Byte hCT[16*numBlock];//블록크기*블록개수
Byte hRK[16*numRnd];//블록크기*라운드수

/* 평문, 라운드 키 설정 */
Byte *dPT, *dRK, *dCT;//GPU용 포인터 선언
//평문 데이터 영역 확보
cudaMalloc((void**)&dPT, 16*numBlock);
//암호문 데이터 영역 확보
cudaMalloc((void**)&dCT, 16*numBlock);
//라운드 키 데이터 영역 확보
cudaMalloc((void**)&dRK, 16*numRnd);
// 평문, 라운드 키 복사(host PC -> 그래픽 카드)
cudaMemcpy(dPT, hPT, 16*numBlock,
                    cudaMemcpyHostToDevice);
cudaMemcpy(dRK, hRK, 16*numRnd,
                    cudaMemcpyHostToDevice);
//커널 구성
dim3  blk(tX, tY);//(tX*tY)개 쓰레드로 블록구성
dim3  grid(bX, bY);//(bX*bY) 블록의 그리드구성
// 커널 호출(암호화 수행)
blockcipher_kernel<<<grid,blk>>>(dSbox,
                                dPT, dRK, dCT);
//암호화 결과 전송(그래픽 카드 -> 호스트 PC)
cudaMemcpy(hCT, dCT, 16*numBlock,
                    cudaMemcpyDeviceToHost);
//그래픽 메모리 해제
cudaFree(dSbox);cudaFree(dPT);
cudaFree(dCT); cudaFree(dRK);
```

이제 그래픽 카드에서 실제로 암호화를 수행하는 커널 함수를 살펴보자. 커널의 구성에 따라 각 쓰레드는 그리드내의 블록 좌표 (bX,bY)와 블록내 쓰레드 좌표 (tX, tY)를 고유 번호로 사용할 수 있다. 이 고유번호에 따라 각 쓰레드에 서로 다른 작업을 할당할 수 있다. 커널에서 그리드의 구성을 참조하기 위해 사전에 준비된 변수(built-in variable)들이 있다. 이 변수들은 모두 dim3의 데이터 형식을 가진다. gridDim과 blockDim은 각각 그리드와 블록의 크기를 저장하고, blockIdx와 threadIdx는 각각 블록 좌표와 쓰레드 좌표를 제공한다.

```
/* 그래픽 카드(GPU)에서 실행되는 소스 코드 */
__global__ void //커널은 항상 golbal void로 선언
 blockcipher_kernel(u32 *dSbox, Byte *dPT,
                    Byte *dRK, Byte *dCT)
{
//쓰레드의 좌표 구한다.
 int tx=threadIdx.x; int ty= threadIdx.y;
 int bx=blockIdx.x;  int by = blockIdx.y;
 thread_idx = (tx*blockDim.y + ty);

//Sbox, 라운드 키 설정(global -> shared 메모리)
 __shared__ u32 Sbox[256];//공유 메모리에 선언
 for(i=0;
      i<256/(blockDim.x*blockDim.y); i++)
 {
  byte_idx=i*(blockDim.x*blockDim.y)
                          +thread_idx;
  Sbox[byte_idx] = dSbox1[byte_idx];
 }
 __syncthreads(); //쓰레드의 작업을 동기화 한다.
/* 라운드 키도 같은 방법으로
          on-chip shared 메모리로 복사한다 */

//암호화에 사용할 평문, 암호문, 라운드 키 포인터
 Byte *PT_ptr; Byte *CT_ptr, *RK_ptr;
//쓰레드별 데이터 위치 결정
 PT_ptr = dPT + thread_idx*16;
 CT_ptr = dCT + thread_idx*16;

/* 암호화 수행(암호화 코드를 여기에 가져온다)  */
}
```

일반적으로 PC에서 블록암호를 소프트웨어로 구현하는 방식은 8비트 참조구현과 32비트 최적화 구현으로 나누어진다. CUDA를 이용하는 경우에도 이 두 가지 방식을 모두 적용할 수 있다.

128비트 블록암호에 대하여, 16개의 쓰레드가 각각 한 바이트씩을 처리하며 공동으로 한 블록의 암호화에 참여하는 방식으로 구현하는 것은 8비트 구현에 대응된

다. 이 경우 16개의 쓰레드가 한 블록이 되는 커널을 구성하고 각 블록은 암호화에 필요한 Sbox와 라운드 키를 온칩 공유 메모리에 복사한다. 각 쓰레드는 담당하는 바이트에서 Sbox 연산을 수행하고, 다른 쓰레드에 대응되는 바이트를 공유 메모리로부터 참조하며 해당 바이트의 암호화를 수행한다. CPU를 이용한 소프트웨어 구현의 경우 순차적인 16번의 Sbox 참조를 피할 수 없지만, GPU에서는 이를 병렬적으로 한번에 처리한다. 하지만, 공유 메모리에 대한 랜덤 엑세스가 불가피하여 각 쓰레드 사이에 동일한 메모리 참조로 지연이 발생한다. 실제로 이러한 8비트 GPU 암호화는 CPU의 성능에 미치지 못한다.

CPU에서 32비트로 최적화된 구현을 각 쓰레드에서 그대로 사용하는 방식으로 블록암호를 구현할 수도 있다. 이 경우 공유 메모리의 사용량이 많아지게 되어 하나의 멀티 프로세서에 탑재할 수 있는 쓰레드의 개수는 감소한다. 하지만, 각 쓰레드가 독립적으로 암호화를 수행하기 때문에 데이터의 상호 참조로 인한 지연은 거의 발생하지 않는다. 또한, 앞에서 설명한 CUDA 프로그램의 틀에서 암호화 수행 부분만 CPU 프로그램을 인용하여 구현할 수 있기 때문에 구현 난이도도 매우 낮다. 암호화의 속도는 블록암호와 GPU의 종류에 따라 다르지만, CPU보다 5배 이상의 고속처리가 가능하다.

Ⅳ. 구현 결과 분석

4.1 속도 측정 및 결과 분석

GPU에서 수행되는 프로그램의 속도를 측정하는 방법으로 CUDA에서 제공하는 타이머를 사용할 수 있다. CPU 프로그램에서 소요시간을 클럭으로 측정할 때 사용하는 __rdtsc()보다는 정밀도가 떨어지지만, clock() 함수 정도의 정확도는 보장한다. 다음은 시간을 측정하는 소스 코드의 일부이다.

```
unsigned int timer = 0;
cutCreateTimer(&timer);
cutStartTimer(timer);
/* (시간을 측정할 CUDA 서브루틴 호출) */
cutStopTimer(timer);
printf("%f ms \n", cutGetTimerValue(timer));
cutDeleteTimer(timer);
```

암호화 속도를 비교하기 위하여, 호스트 PC에서 그래픽 카드의 글로벌 메모리에 데이터를 전송하는 시간은 제외하였다. 즉, 글로벌 메모리에 저장된 평문 데이터를 암호화하여 다시 글로벌 메모리의 지정된 위치에 넣은 데 소요되는 시간으로 처리속도를 계산하고 비교하였다.

속도 측정 환경으로 Core 2 Duo 6600 CPU, 1GB 메모리를 가진 PC에 NVIDIA Geforce 8800을 사용하였다. AES와 ARIA는 8비트, 32비트 구현을 모두 실험하였고, DES의 경우는 8비트로 분할하는데 무리가 있어 하나의 쓰레드가 독립적인 DES 암호화를 수행하는 방식으로 처리하였다. 반복 측정하여 평균 속도를 계산한 결과는 [표 3]과 같다. 일반적인 CPU 소프트웨어 구현 시 ARIA는 AES보다 많은 라운드 수를 가지고 있기 때문에 속도가 더 느리지만, GPU 프로그램에서는 더 빠르게 구현이 가능하였다. 그 이유에 대해서는 다음 절에서 분석하기로 한다.

속도 측정 환경은 NVIDIA 8800GTS와 8800GTX로 각각 12개와 16개의 멀티 프로세서를 가지고 있다. 구현 결과는 [표 3]과 같으며, 프로세서의 개수와 메모리 대역폭을 고려하면, 프로세서의 개수에 거의 정비례하는 속도 향상을 보임을 알 수 있다.

〔표 3〕 CUDA를 사용한 블록암호의 구현 속도

알고리즘	실행환경/암호화 속도	
	8800GTS	8800GTX
AES	3.1 Gbps	4.5Gbps
ARIA	4.8 Gbps	7.0Gbps
DES	1.6 Gbps	2.8Gbps

PC에서 소프트웨어로 구현된 블록암호의 속도는 대개 1Gbps 이하의 성능을 나타낸다[14]. 최신 CPU인 인텔의 Core2 quad(2.4GHz)에서도 C로 구현된 AES와 ARIA는 각각 1.1Gbps, 625Mbps에 그치고 있다. 따라서 GPU를 이용한 PC에서의 소프트웨어 구현은 CPU를 이용하는 기존의 속도를 능가하는 결과를 보인다. AES를 하드웨어로 구현하는 경우 23Gbps[15]까지 가능하다는 결과가 있으며, ARIA도 20Gbps급 설계 기법이 알려져 있다. 전용 블록암호 칩셋을 사용하는 것이 구현속도나 경제적인 측면에서 우수하나, GPU를 이용하면 PC에서 소프트웨어 구현으로도 전용 칩셋에 근접한 결과를 얻을 수 있음을 알 수 있다. 특히, 여러 개의

그래픽 카드를 PC에 연결하면, 전용 칩셋 이상의 속도를 기대할 수 있으며, 필요에 따라 다양한 알고리즘을 적용할 수 있는 유연성을 가질 수 있다.

4.2 최적화 방안

NVIDIA의 G80 계열 GPU는 [표 2]에 설명한 바와 같이 각 멀티 프로세서마다 8천 여개의 레지스터와 16KB의 공유 메모리가 있다. 이를 하나 혹은 그 이상의 쓰레드 블록이 공유하는 것을 생각하면 GPU 프로그래밍에서는 부족한 메모리를 효율적으로 활용하는 것이 매우 중요하다. CUDA 프로그래밍에서는 작은 크기의 쓰레드를 많이 사용할 것을 권장하고 있으며, MS 윈도우와는 달리 쓰레드 간의 전환은 매우 빠르게 일어난다(4 cycles 소요). 각 멀티 프로세서에 8개의 스트림 프로세서가 있고, 4단계의 파이프 라인 구조를 가지기 때문에 하나의 멀티 프로세서에는 동시에 최대 32개의 쓰레드가 연산에 참여할 수 있다. 쓰레드에서 데이터의 입출력이 지연되는 동안 다른 쓰레드가 실행되는 것을 감안하면 1,000개 이상의 쓰레드로 커널을 구성해야 GPU의 연산 능력을 최대로 활용할 수 있다. NVIDIA는 경험적인 수치로써, 최소한 64개 이상의 쓰레드로 블록을 구성할 것을 요구하고 있으며, 192 또는 256 개 이상의 쓰레드를 권장하고 있다.

〔표 4〕 블록암호 구현에 사용된 메모리 비교

구현방식	알고리즘	shared memory	register	local memory
8 비트	AES	1,016 바이트	12개	20 바이트
	ARIA	1,332 바이트	14개	7 바이트
32 비트	AES	5,168 바이트	63개	176 바이트
	ARIA	4,348 바이트	19개	0 바이트
	DES	2,080 바이트	17개	128 바이트

[표 4]는 블록암호 구현에서 사용된 메모리의 크기를 나타낸다. 이 값은 컴파일시 '-cubin' 옵션으로 얻을 수 있다. 이 표에서 상대적으로 가장 소모가 많은 자원은 Sbox를 저장하는 공유 메모리이다. 32비트 구현에서 ARIA가 AES를 능가하는 이유는 고속구현시 필요한 Sbox의 개수가 AES보다 하나 적기 때문이다. 즉, 공유 메모리 사용량의 차이로 멀티 프로세서에 올려놓을 수 있는 쓰레드 블록의 개수가 달라지기 때문이다. 또한, 최종 라운드를 다른 라운드와 달리 구현하는 AES보다 프로그램의 흐름을 간단히 구현할 수 있어 레지스터만으로 구현이 가능하여, 레지스터 부족으로 로컬 메모리까지 사용한 AES보다 시간 지연(latency)을 줄일 수 있었던 것도 속도에 영향을 주었다. DES의 경우 많은 비트 연산이 필요하여 공유 메모리 점유는 적었으나 실행 속도는 느린 결과를 얻었다.

가장 중요한 최적화는 메모리의 관리이지만, 다른 요소들도 많게는 10배까지 속도에 영향을 줄 수 있음이 알려져 있다. 작은 크기의 정수 곱셈의 경우 __mul24() 라는 24비트 곱셈을 사용하면 4배 빠르게 계산된다. 또한 메모리의 부족 때문에, 여러 번 중복 사용되는 연산의 중간 결과를 저장하여 활용하는 것보다 필요시 다시 계산하는 경우가 더 빠를 수도 있다.

V. 결 론

GPU의 계산 능력이 CPU를 앞서감에 따라 시작된 GPGPU 연구는 2007년 CUDA 라이브러리의 발표로 크게 확산되고 있다. CUDA는 비록 NVIDIA의 GPU에서만 동작한다는 제약이 있지만, 그 성능과 프로그래밍의 편이성 때문에 병렬 연산이 필요한 과학 기술 분야에서 크게 환영받고 있다. 본고에서는 CUDA 라이브러리를 이용한 블록암호의 구현을 소개하였다. 대부분의 블록암호는 기존 CPU용 소스 코드로부터 쉽게 구현할 수 있으며, 성능도 CPU를 크게 능가하는 결과를 얻었다. 구현 성능은 OpenGL 확장기능으로 구현한 NIDIA GPU 상에서 알려진 가장 빠른 결과인 Yamanouchi[10]의 구현과 동등한 수준이다. 하지만, 그래픽 처리에 대한 이해가 없어도 쉽게 GPU상에서 블록암호를 구현할 수 있다는 장점이 있다.

정보보호 분야에서 GPU의 사용은 부가적인 장점을 많이 제공한다. GPU에서 블록암호를 구현한다는 것은 CPU의 점유율을 낮추게 되어 암호의 사용으로 인한 오버헤드를 크게 줄일 수 있음을 의미한다. 현재의 부채널 공격도 GPU 메모리에 까지는 도달하지 못하고 있으며, GPU를 난수발생기나 스트림 암호에 사용하는 것도 좋은 응용이 될 수 있다. 논문에서 제시한 CUDA 프로그램의 기본 프레임 웍은 기존의 CPU용 블록암호 소스 코드를 GPU용으로 변환하는 도구로 활용될 수 있으며 암호구현 분야에서 GPU의 사용을 촉진할 수 있을 것으로 기대된다.

참고문헌

[1] M. Houston, "GPGPU : General-Purpose Computation on Graphics Hardware", Course at SIGGRAPH 2007.

[2] The GPGPU Resources and Forums, http://www.gpgpu.org/.

[3] The first GPGPU workshop, Proceedings of the first Workshop on General Purpose Processing on Graphics Processing Units, http://www.ece.neu.edu/GPGPU, 2007.

[4] Astro GPU, Workshop on General Purpose Computation on Graphics Processing Units in Astronomy and Astrophysics, http://astrogpu. org, 2007.

[5] GPUbench project, http://graphics.stanford.edu/projects/gpubench/.

[6] D. Shreiner, M. Woo, J. Neider, T. Davis, OpenGL Programming Guide : The Official Guide to Learning OpenGL. Version 2, Addison Wesley, 2005.

[7] D. L. Cook, J. Ioannidis, A. D. Keromytis, J. Luck "CryptoGraphics : Secret Key Cryptography Using Graphics Cards", CT-RSA, Springer LNCS 3376, 2005.

[8] D. L. Cook, A. D. Keromytis, Cryptographics : Exploiting Graphics Cards for Security, Advances in Information Security series, Springer, 2006.

[9] O. Harrison, J. Waldron, "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units", CHES, Springer LNCS 4727, 2007.

[10] T. Yamanouchi, "AES Encryption and Decryption on the GPU", GPU Gems 3, NVIDIA, 2007.

[11] J. Yang, J. Goodman, "Symmetric Key Cryptography on Modern Graphics Hardware", ASIACRYPT, Springer LNCS 4833, 2007.

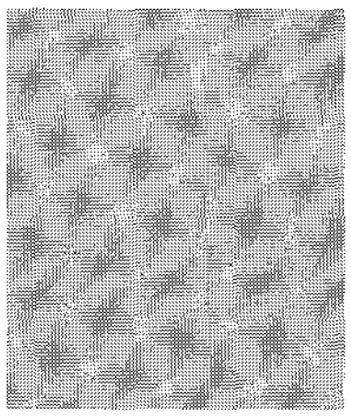
[12] NVIDIA CUDA Homepage, http://developer.nvidia.com/object/cuda.html.

[13] D. Göddeke, GPGPU - Basic Math Tutorial, Technical report No. 300, Fachbereich Mathematik, Universität Dortmund, 2005.

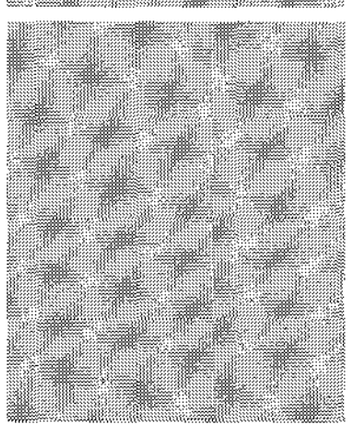
[14] 장환석, 이호정, 구본욱, 송정환, "64비트 마이크로프로세서에 적합한 블록암호 ARIA 구현방안", 정보보호학회논문지 제16권 3호, 2006.

[15] J. Zambreno, D. Nguyen, A. Choudhary, Exploring area/delay tradeoffs in an AES FPGA implementation, Proc. 14th Int. Conf. Field-Programmable Logic and its Applications, FPL 2004.

〈著者紹介〉



염 용 진 (Yongjin Yeom) 정회원
1991년 2월 : 서울대학교 수학과 졸업
1994년 2월 : 서울대학교 수학과 석사
1999년 2월 : 서울대학교 수학과 박사
2000년 4월~현재 : ETRI부설연구소 선임연구원
<관심분야> 암호이론, 정보보호



조 용 국 (Yongkuk Cho) 정회원
2000년 2월 : 한양대학교 수학과 졸업
2002년 2월 : 한양대학교 수학과 석사
<관심분야> 암호이론, 정보보호