

# 목차

<b>1 개요</b>	<b>1</b>
1.1 연구 동기 . . . . .	1
1.2 논문 기여 . . . . .	2
1.3 논문 구성 . . . . .	2
<b>2 GPGPU와 Curve25519 기반 ECDH</b>	<b>3</b>
2.1 GPGPU . . . . .	3
2.2 병렬처리 . . . . .	3
2.3 병렬처리를 위한 GPGPU 활용 기술 . . . . .	5
2.3.1 CUDA . . . . .	5
2.3.2 OpenCL . . . . .	8
2.4 Curve25519 기반 ECDH . . . . .	12
2.4.1 타원곡선 암호화 . . . . .	12
2.4.2 이산대수문제 . . . . .	13
2.4.3 ECC Operation . . . . .	14
2.5 Curve25519 . . . . .	20
2.5.1 Curve25519의 특징 . . . . .	21
2.6 ECDH . . . . .	22
<b>3 GPGPU 성능 향상 기법</b>	<b>24</b>
3.1 다중 쓰레딩 . . . . .	24
3.2 루프 언롤링(Loop Unrolling) . . . . .	24
3.3 코드 직렬화 . . . . .	26

3.4	다중 병렬화 . . . . .	27
3.5	메모리 정렬 . . . . .	28
3.6	연산 정렬 . . . . .	29
<b>4</b>	<b>GPGPU상 Curve25519 구현 및 성능 향상 방법</b>	<b>31</b>
4.1	타겟 GPU 및 구현 환경 . . . . .	31
4.2	curve25519 구현 . . . . .	31
4.2.1	필드 곱셈 병렬화 . . . . .	32
4.2.2	동고메리 double and add 병렬 방법 . . . . .	36
4.3	다중쓰레딩 . . . . .	36
4.4	루프 언롤링 . . . . .	36
4.5	코드 직렬화 . . . . .	39
<b>5</b>	<b>결론</b>	<b>45</b>
<b>A</b>	<b>약어</b>	<b>48</b>
<b>B</b>	<b>초록</b>	<b>48</b>

## 표 목차

1	NIST Curve Dmoain	13
2	RSA와 ECC 암호 강도 비교	13
3	$EC(y^2 = x^3 + ax + b)$ 식	14
4	nP 연산	15
5	필드 곱셈 병렬화 속도 비교	41
6	몽고메리 double and add 병렬화 연산 횟수 비교	41
7	GTX 1050Ti에서 쓰레드개수별 400회 Curve25519 수행 시간	42
8	쓰레드개수별 400회 Curve25519 수행 시간에 대한 처리량	42
9	Unrolling 미적용 수행 시간	43
10	Unrolling 적용 수행 시간	43
11	Unrolling 적용, 미적용 간 성능 향상	43
12	코드 직렬화 비율	44
13	코드 직렬화 미적용코드 수행 시간	44
14	코드 직렬화 적용코드 수행 시간	44
15	코드 직렬화 적용코드 수행 시간 비교	44

## 그림 목차

1	CPU and GPU . . . . .	3
2	암달의 법칙 . . . . .	4
3	구스타프슨의 법칙 . . . . .	5
4	Cuda Thread . . . . .	6
5	Cuda Memory . . . . .	7
6	OpenCL의 구조 . . . . .	9
7	OpenCL의 그리드 . . . . .	10
8	타원곡선 . . . . .	12
9	$y^2 = x^3 + 4x^2 + x$ 몽고메리 타원곡선 . . . . .	21
10	Curve25519상 몽고메리 사다리 . . . . .	22
11	Curve25519를 사용하는 ECDH 키교환 . . . . .	23
12	싱글 쓰레드와 멀티 쓰레딩 . . . . .	25
13	다중 병렬화 . . . . .	28
14	연산 정렬 . . . . .	29
15	연산 정렬 . . . . .	30
16	몽고메리 double and add 연산 순서 . . . . .	33
17	필드 곱셈 . . . . .	35
18	필드 곱셈 병렬화 . . . . .	35
19	몽고메리 double and add 연산 병렬화 . . . . .	37
20	쓰레드 개수별 수행시간 그래프 . . . . .	38
21	쓰레드 개수별 처리량 그래프 . . . . .	38

# 1 개요

## 1.1 연구 동기

GPGPU(General Purpose Graphic Processing Unit)는 기존 그래픽처리를 빠르게 하기위한 그래픽 처리장치(Graphic Processing Unit, GPU)를 기존 CPU처럼 사용하는 것이다. 이러한 GPGPU는 기존의 그래픽처리뿐만 아니라 딥러닝, 암호연산 등 다양한 분야에서 응용되고 있다. 이러한 GPU를 활용하기 위해 NVIDIA사의 CUDA, 크로노스 그룹(Khronos Group)에서 제공하는 OpenCL, OpenGL 그리고 차세대 GPU API인 Vulkan과 같은 다양한 라이브러리들이 제공되고 있으며 이와 관련된 연구는 계속 진행 중이다. 이러한 GPGPU는 암호 분야에서도 활발히 사용되고 있으며, 대칭키 암호, 해시 함수, 비대칭키 암호에 대하여 대규모처리, 병렬처리를 이용한 가속화 기법 등 다양한 연구가 진행되고 있다.

이전까지 대중적으로 사용되는 타원곡선 암호는 미국 국립표준 기술연구소(National Institute of Standards and Technology, NIST)에서 제시하는 NIST-P256이었으나, 2013년 미 국방성(National Security Agency, NSA)에서 NIST P256을 사용하는 Dual\_EC\_DRBG표준에 백도어를 심었다는 주장이나오며 알고리즘적 결함을 발견하게 되었다. 이에 RSA Security는 Dual\_EC\_DRBG을 더이상 사용하지 않도록 권고하였다. 이후 이를 대체하기 위해 SafeCurve 프로젝트가 진행되었으며, 이 중 하나로 2005년에 다니엘 번스타인(Daniel J. Bernstein)이 제시한 Curve25519가 주목받고 있다. 차세대 곡선으로 주목받고 있는 Curve25519는 앞으로 다양한 분야에서 응용될 것이고 특히 네트워크 통신 시 안전한 통신 채널 성립을 위한 보안 프로토콜(Secure Socket Layer, SSL)에서 많은 사용이 예상된다. 보안 프로토콜은 일대일 통신 채널을 생성하여 많은 사용자가 접속하게 되는 경우 매번 해당 키를 이용하여 암.복호화를 해주어야 한다. 하지만 이러한 많은 접속자의 암.복호화 처리를 CPU에서 혼자 하는 경우 다른

동작에 대한 성능이 하락할 수 있어 GPGPU와 같은 대규모 병렬처리장치를 이용하여 암.복호화 하는 경우 효율적으로 빠르게 많은 데이터를 암.복호화 할 수 있다.

## 1.2 논문 기여

기존 NIST P256에대한 연구에 비하여 Curve25519에대해 진행된 연구는 상대적으로 매우 적은 편이다. 특히 GPGPU를 활용한 성능향상 기법에 대한 연구는 ECDH 상 스칼라 곱(scalar Multiplication)에 대한 병렬처리에 대한 연구만이 진행되어 연구가 많이 필요한 상황이다. 본 논문에선 스칼라 곱 뿐만 아니라 필드 상 곱셈 연산과 같은 세부적인 연산까지 전체연산에 대하여 GPGPU의 특성을 활용한 성능향상기법을 적용한다.

## 1.3 논문 구성

본 논문은 총 5장으로 구성된다. 2장에서는 본 연구를 위한 배경지식으로 병렬처리와 GPGPU, Curve25519 기반 ECDH 타원곡선 암호화에 대해서 설명한다. 3장에서는 GPGPU에 적용 가능한 속도향상기법에 대하여 알아보며. 4장에서는 구현 결과물과 성능 분석을 한다. 마지막으로 5장에서는 본 논문에 대한 결론을 내린다.

## 2 GPGPU와 Curve25519 기반 ECDH

### 2.1 GPGPU

GPGPU(General Purpose Graphic Processor Unit)는 기존 그래픽처리를 위하여 개발된 그래픽 처리 장치를 여러 대의 CPU처럼 사용하는 기술이다. GPU의 ALU는 CPU의 그것보다 적은 기능을 가지지만, 수백 수천 개의 프로세서를 가지고 있어 동시에 대량의 데이터를 처리할 수 있는 특징을 가지고 있다.



Figure 1: CPU and GPU

### 2.2 병렬처리

병렬처리(Parallel Processing)는 동시에 많은 연산을 수행하는 방법으로 크고 복잡한 문제를 작게 나누거나 동시에 많은 연산을 수행하여 특정한 작업을 가속화 하는 방법이다. 병렬처리는 동시에 여러 종류의 데이터를 처리함으로써 동기화 문제 등 다양한 이슈가 존재한다.

병렬화를 통한 속도는 선형적으로 증가하며 프로세서의 개수가 증가함으로써 이에 비례하여 성능이 향상된다. 이러한 병렬화를 통한 속도 향상을 계산하는 방법으로 암달의법칙과 구스타프슨의 법칙이 있다. 암달의 법칙은 시스템 개선 시 최대성능 향상의 최대치를 계산하는 방법으로 다음 식과 같다.

$$S = \frac{1}{1 - P}$$

전체 작업 중 병렬 가능한 구간을  $P$ 라고 하였을 때 성능향상은  $S$ 와 같다. 만약 75%의 병렬 가능한 구간이 있다면 최대 성능은 4배까지 상승할 수 있다.

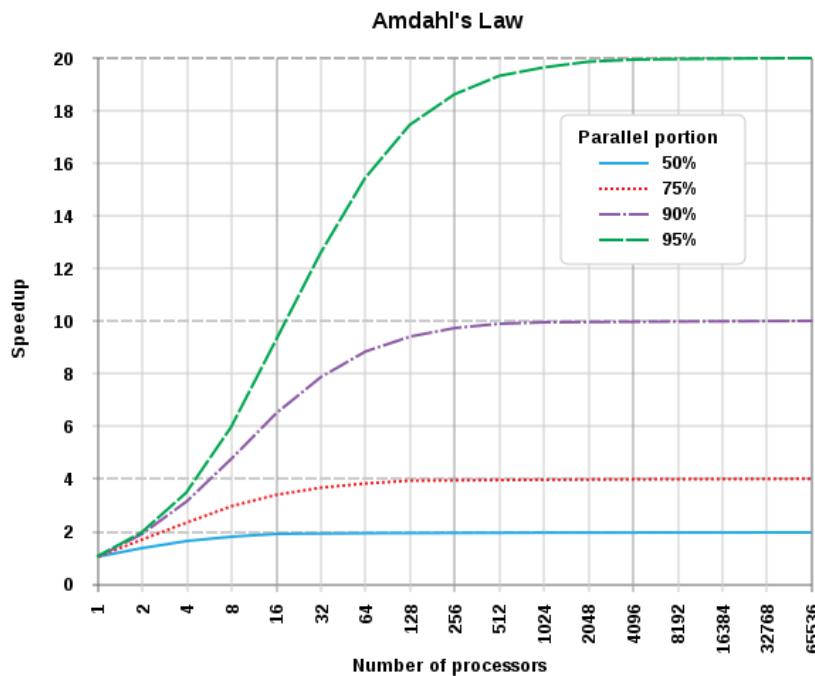


Figure 2: 암달의 법칙

구스타프슨의 법칙은 병렬처리는 대용량 데이터처리에 효과적임을 보여준다.  $P$ 는 프로세서의 수,  $a$ 는 병렬화되지 않는 구간의 비율,  $S$ 는 성능향상을 나타낸다.

$$S(P) = P - a \times (P - 1)$$

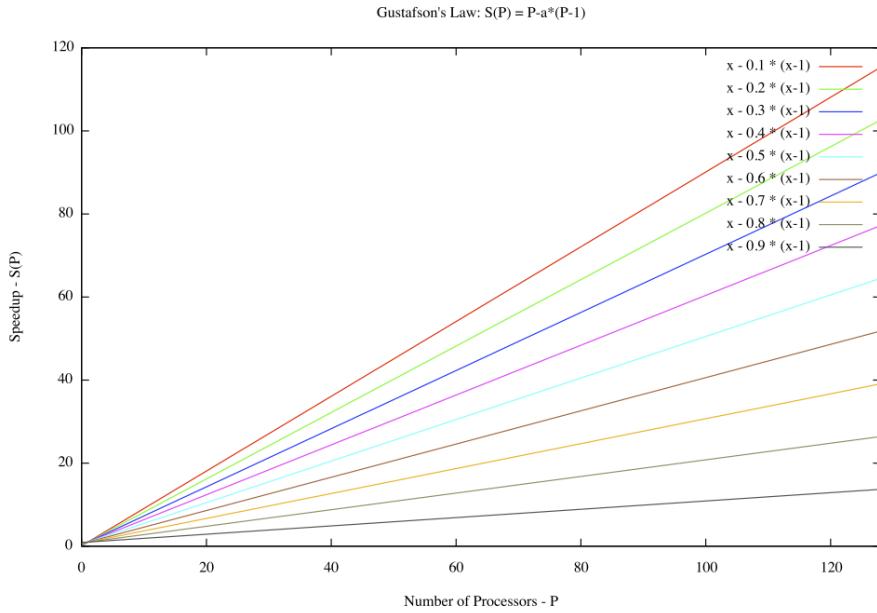


Figure 3: 구스타프슨의 법칙

### 2.3 병렬처리를 위한 GPGPU 활용 기술

GPGPU를 다루는 기술들로는 CUDA, OpenCL, OpenGL, OpenACC, Vulkan 등이 있으며, 현재도 다양한 라이브러리들이 다양한 목적으로 개발되고 있다. 이중 CUDA와 OpenCL이 대표적인 GPGPU 활용 라이브러리로 사용되고 있다.

#### 2.3.1 CUDA

CUDA는 NVIDIA에서 만든 병렬 컴퓨팅 플랫폼으로 쿠다 코어를 가진 GPU의 명령어를 사용할 수 있도록 만들어주는 라이브러리로써 NVIDIA의 쿠다코어가 장착된 GPU만 사용이 가능하다. 2006년 최초로 소개되었으며, C/C++ 뿐만 아니라 python C# 등 다양한 언어를 지원하고 있다.

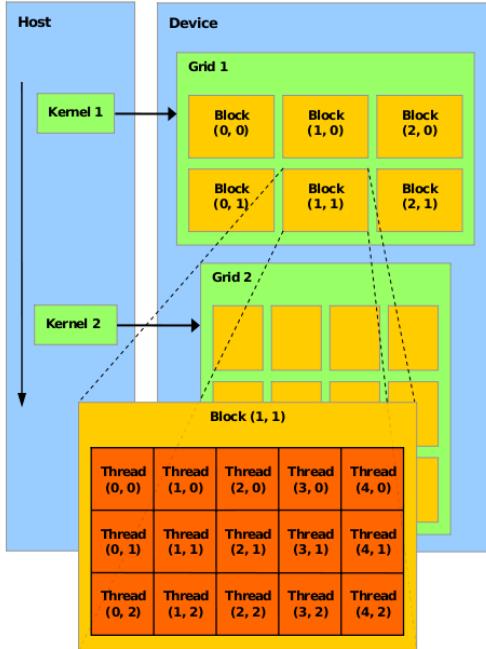


Figure 4: Cuda Thread

CUDA 의 쓰레드모델은 그림-4와 같다. Host에서 실행한 커널은 디바이스 함수를 호출할 수 있으나, 디바이스 내의 함수에선 호스트 함수를 호출할 수 없다. 하나의 디바이스에선 하나의 커널만 존재할 수 있으며, 각각의 커널은 여러 개의 쓰레드를 동작시킨다. 디바이스 내엔 각각이 블록과 각 블록 내에 쓰레드가 있으며, 자신의 위치를 BlockIdx와 ThreadIdx를 이용하여 알 수 있다. GPU는 메모리가 계층 구조로 되어 있으며 글로벌 메모리(Global Memory), 공유 메모리(Shared Memory), 로컬메모리(Local Memory), 레지스터(Registers)의 4가지 메모리로 구성된다. 글로벌 메모리는 용량이 가장 크며, 커널 단위로 접근이 가능하지만, 속도가 느린다. 공유메모리는 블록 단위에서 접근이 가능하며 빠르게 읽고 쓸 수 있다. 로컬메모리와 레지스터는 쓰레드 단위로 접근할 수 있으며, 가장 빠른 접근속도로 읽고 쓸 수 있지만 크기가 작다. 표-5은 이러한 메모리 구조를 나타낸다.

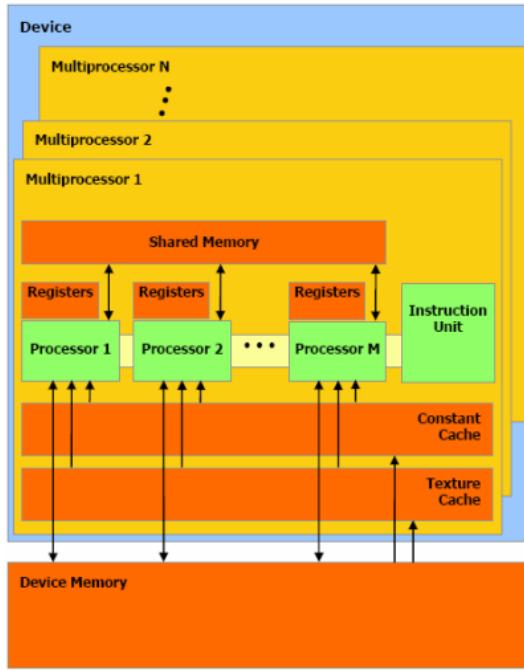


Figure 5: Cuda Memory

쿠다 코드는 글로벌코드와 디바이스코드로 구분되어 있으며 글로벌은 호스트 혹은 다중 병렬처리 시 호출되는 커널 코드이며, 디바이스코드는 디바이스에서만 호출 가능한 디바이스 함수이다. 커널 함수는 글로벌 함수로써 호스트에서만 실행될 수 있지만, 다중 병렬처리를 지원하는 디바이스에선 커널 함수가 아닌 디바이스 코드에서도 커널 함수를 호출 할 수 있다. 커널 함수에 전달하기 위해선 디바이스의 메모리에 데이터를 전달하고, 결과값들 받을 버퍼를 할당해야 한다. 버퍼는 'cudaMemalloc'으로 할당 할 수 있으며 호스트의 메모리 데이터를 'cudaMemcpy'를 이용하여 복사한다.

$$idx = blockIdx.x * blockDim.x + threadIdx.x$$

커널 코드는 재귀호출이 불가능하며, 호스트로부터 호출되어 디바이스에서 최초로 동작하는 함수이다. 커널은 쓰레드 단위로 동작하며 자신의 위치를 알기 위해선

그리드 좌표와 쓰레드 번호를 이용해서 알아낼 수 있다. 디바이스코드는 호스트에 접근할 수 없으며, static 변수를 선언할수 없으며, 호스트에서도 호출할 수 없다.

다음은 커널 코드의 동작에 대한 예시이다.

```
--global-- void kernel( int* a, int* b, int n){  
    idx = blockIdx.x * blockDim.x + threadIdx.x;  
    /* Same operation  
    for ( int i = 0 ; i < n ; i++)  
        a[ i ] = a[ i ] + b[ i ];  
    */  
    if ( idx < n )  
        a[ idx ] = a[ idx ]+b[ idx ];  
    return ;  
}
```

### 2.3.2 OpenCL

OpenCL은 크로노스 그룹에서 만들고 관리하는 병렬 컴퓨팅 플랫폼으로 GPU뿐만 아니라 CPU, Accelerator 등 다양한 플랫폼에 대하여 병렬처리를 가능하게 하는 기술이다. OpenCL은 CUDA 와 같이 커널프로그램을 작성하여 디바이스를 동작시키는 것은 동일하지만, CUDA 와는 달리 CUDA 코어가 아니더라도 CPU, FPGA, GPGPU 및 하드웨어 가속기 등 다양한 플랫폼에서 동작할 수 있다.

OpenCL의 구조는 그림-6와 같다. 호스트는 OpenCL명령을 전달하는 호스트 프로그램으로 주로 CPU가 되며 연산 장치(Compute Device)는 GPGPU, CPU, FPGA

등 타겟 디바이스가 된다. 연산유닛(Compute Unit)은 디바이스의 한 연산 블록이며 CUDA 의 멀티프로세서하나와 같다. 처리장치(Processing Element, SM)는 실제로 코드가 수행되는 쓰레드 하나이다. 메모리 또한 쿠다와 거의 비슷한 구조를 가지며 글로벌 메모리, 공유 메모리, 로컬 메모리, 레지스터의 4계층의 구조를 가지며 각각의 메모리에 대하여 명시적으로 가리킬 수 있다.

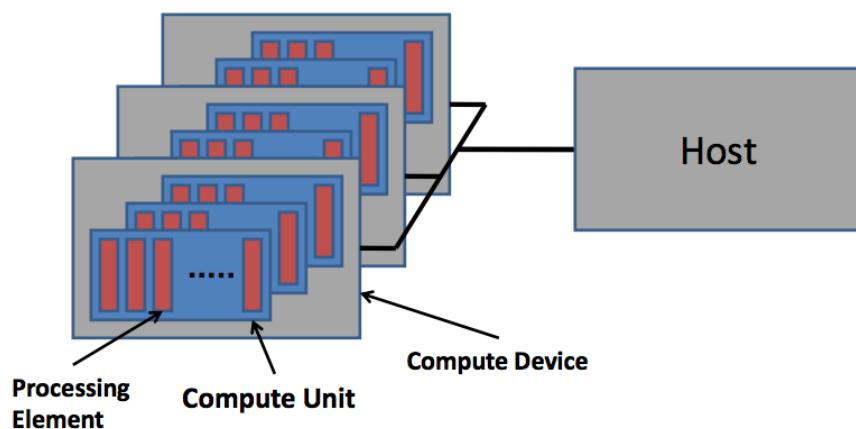


Figure 6: OpenCL의 구조

OpenCL에서 그리드는 그림-7와 같다. 큰 그룹은 워크그룹(Work group)이며 각 그룹은 글로벌 메모리와 공유 메모리를 공유할수 있다. 워크그룹 내에선 각쓰레드는 워크 아이템이 되며 이는 병렬처리의 최소단위이다. 쓰레드가 자신의 위치를 찾는 식은 다음과 같다

$$idx = get_{local\_size}(dim) * get_{group\_id}(dim) + get_{local\_id}(dim)$$

자신의 위치는 글로벌 아이디와 같다.

다음은 OpenCL 커널코드의 예시이다

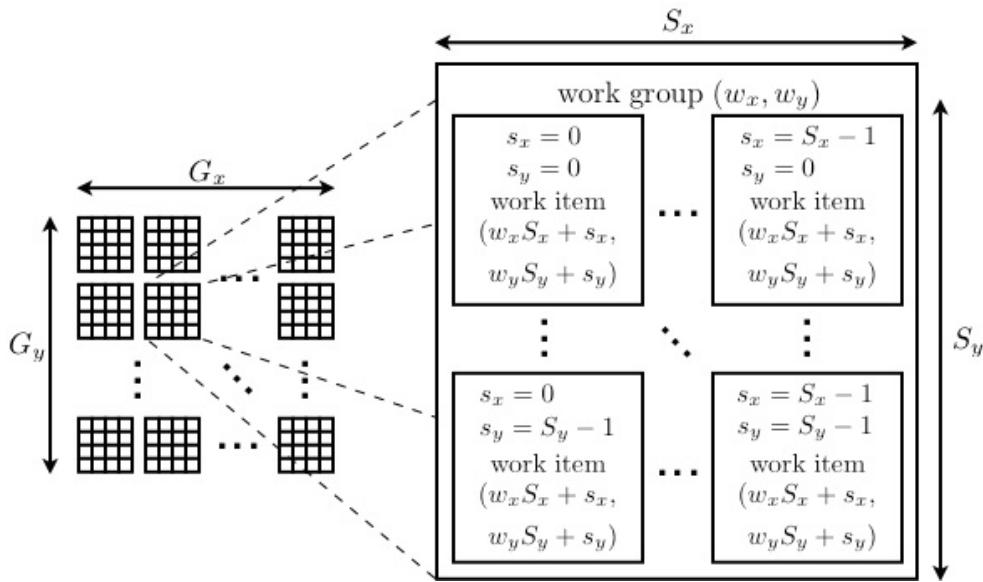


Figure 7: OpenCL의 그리드

```

__kernel void dataParallel( __global float* A, __global float* B, __global float* C)
{
    int base = 4*get_global_id(0);
    C[base+0] = A[base+0] + B[base+0];
    C[base+1] = A[base+1] - B[base+1];
    C[base+2] = A[base+2] * B[base+2];
    C[base+3] = A[base+3] / B[base+3];
}

```

커널코드는 `__kernel`이라는 명시자로 시작하여 커널코드임을 알려주며, 병렬처리의 도입부가된다. 이후부터 디바이스내에서 동작하게되어 호스트의 자원은 사용할수 없게 된다. OpenCL은 입출력 속도가 빠른 로컬메모리를 사용하는 변수에 대하여 명시적으로 선언 할수 있으며, 이때 로컬메모리의 크기를 고려하여 로컬메모리를 사용하는

변수의 크기를 정하여야 한다. 로컬메모리의 크기가 크지 않기 때문에 자주 사용하는 배열을 로컬메모리에 배정하여 성능을 올릴 수 있다.

## 2.4 Curve25519 기반 ECDH

### 2.4.1 타원곡선 암호화

타원곡선 암호화(Elliptic Curve Cryptography, ECC)는 1985년 Neal Koblitz와 Victor Miller가 제안한 알고리즘으로 높은 보안성과 효율성으로 RSA 이후 대표적인 공개키 암호화 시스템으로 자리잡고 있다. 타원곡선 암호화는 Prime fields  $GF(p)$ 와 Finite Fields  $GF(2^n)$ 상에 정의된다. 이러한 타원곡선은  $y^2 + axy + by = x^3 + cx^2 + dx + e$  와 같은 형태이며 이는 그림-8과 같다.

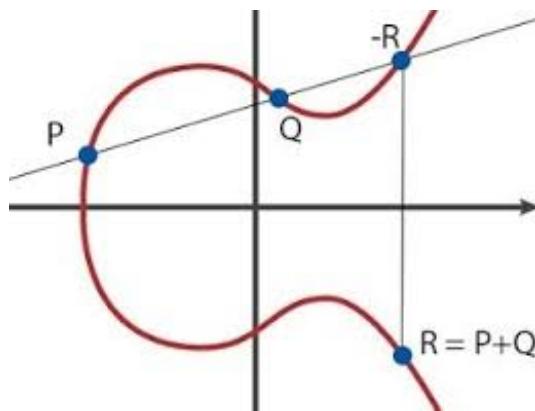


Figure 8: 타원곡선

타원곡선 암호화에선 타원곡선 상의 두 점 P 와 Q를 정하여 두 점을 이었을 때 곡선에 닿는 점(-R)을 x축 대칭한 점(R)을 두 점의 합이라고 하며, 이런 방식으로 계속 더하는 점은 모두 타원곡선 상에 존재하게 된다. 타원곡선 암호화는 이러한 특징을 이용 한 이산대수문제(discrete logarithm problem, DLP)를 기반으로 하며 곡선상의 두 점에 대하여 합연산을 몇 번 했는지 알기 힘든 점을 이용한다. 이러한 암호화 기법은 키 사이즈 256bit에서 AES128과 같은 암호강도를 나타내며, 이는 RSA3072와 같은 암호 강도를 가진다(표-2).

미국 국립표준 기술연구소(National Institute of Standard and Technology, NIST)

에서 타원곡선 암호화시 키 길이별로 추천하는 NIST Curve Domain이 있으며, 이는 표 -1와 같다.

Table 1: NIST Curve Domain

이름	키 길이(bit)	프라임 필드(p)
NIST P-192	192	$2^{192} - 2^{64} - 1$
NIST P-224	224	$2^{192} - 2^{96} + 1$
NIST P-256	256	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
NIST P-384	384	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
NIST P-521	521	$2^{521} - 1$

Table 2: RSA와 ECC 암호 강도 비교

키의 길이	대칭키 암호	동일 암호 강도 RSA 키 길이	동일 암호 강도 ECC 키 길이	RSA와 ECC 키의 길이 비교
56	-	512	112	5:1
80	SKIPJACK22	1024	160	6:1
112	TripleDES	2048	224	9:1
128	AES-128	3072	256	12:1
192	AES-192	7680	384	20:1
256	AES-256	15360	512	30:1

#### 2.4.2 이산대수문제

이산대수(discrete logarithm problem, DLP)문제는 소수  $p$ 가 곱셈에 대해 닫힌 정수군( $Z_n$ )에서  $k$ 제곱을 구할경우  $p^k$ 를 계산한 뒤  $n$ 으로 나눈 나머지를 구하면 된다. 이때 나온 나머지 값을 역으로 올라가  $k$ 를 알아내는 경우는 알아내기가 어려운데 이를 이용한 문제를 이산 대수문제라고한다. 대표적으로 RSA 암호화 방식이 이산대수 문제에 기반하고 있다. 이에 타원곡선 암호화는 ECDLP(Elliptic Curve Discrete Logarithm Problem)에 기반하며, 이는 프라임필드 상에 정의된 커브에서 두 점  $P, Q$ 가 존재할 때,  $lP = Q$ 를 만족하는  $l$ 을 찾기 어려움을 이용하는 것이다.

### 2.4.3 ECC Operation

타원 곡선(Elliptic Curve, EC)에서 연산은 크게 스칼라곱셈(Scalar Multiplication)과 프라임 필드연산(Prime Field Operation)으로 나눌수 있다. 스칼라곱 연산은 타원곡선 상의 두 점을 더할때 사용하는 방식으로 포인트 에디션(Point Addition)과 포인트 더블링(Point Doubling)의 두 개의 연산에 대하여 정의한다. 포인트 에디션은 두 점이 서로 다른 점( $P \neq Q$ )일때 사용하며, 포인트 더블링은 두 점이 같은 점( $P = Q$ )인 경우 사용하는 방식이다. 필드연산은 타원곡선상의 점이 프라임 필드에 정의되어 있기 때문에 값이 P를 넘어가게 되면 필드 안으로 돌려주는 연산을 의미한다.

#### 2.4.3.1 스칼라 곱셈

스칼라 곱셈은 주어진 기저(base point)에 대하여 n배를 해주는 연산을 의미한다. 이를 위에 타원곡선 상에서 정해지는 식이 있으며, 표-3는  $y^2 = x^3 + ax + b$ 에 대한 정의이다.

Table 3:  $EC(y^2 = x^3 + ax + b)$  식

연산	식
영원	$P + O = O + P = P$
덧셈(뺄셈)	$P + (-P) = 0$ , where $-P = (x, x + y)$
$x_3$	$x + 3 = k^2 - (x_1 + x_2)$
$y_3$	$y_3 = -y_1 + k(x_1 - x_3)$
$k(P \neq Q)$	$k = (y_1 - y_2)/(x_1 - x_2)$
$k(P = Q)$	$k = 3x^2 + a/2y$

스칼라 곱셈 연산은 점( $P$ )에 대하여 덧셈을  $n$ 번 해줌으로써  $nP$ 를 구하는 것으로, 위의 식을 기반으로 연산이 가능하다. 예를 들어 점  $P = (x, y)$ 에 대하여 5배를 한다면,  $P$ 를 5번 자기 자신을 더하면 된다(표-4).

이러한 스칼라 곱을 빠르게 수행하기 위한 방법으로 'double-and-add', 'windowed

Table 4:  $nP$  연산

$$nP = P + P + P + P + \dots + P$$

method’, ‘sliding-window method’, ‘wNAF(windowed non-adjacent form) method’, 그리고 ‘montgomery ladder’방법 등이 있다.

---

**Algorithm 1:** double-and-add

---

**Require:**  $P, n$   
**Ensure:**  $P$ : point,  $n$  : 스칼라곱 계수,  $n_i$  :  $n$ 의  $i$ 번째 비트

- 1:  $Q = 0$
- 2: **for**  $i \leftarrow 0 \dots 255$  **do**
- 3:    $Q = point\_double(Q)$
- 4:   **if**  $n_i = 1$  **then**
- 5:      $Q = point\_add(Q, P)$
- 6:   **end if**
- 7: **end for**
- 8: **return**  $Q$

---

double-and-add 방식(알고리즘-1)은  $n$ 의 가장 왼쪽비트(Most Significant Bit, MSB) 부터 확인하여 기존 점을 2배 (left shift) 한뒤, 현재 위치의 비트가 1인 경우 기저( $P$ )를 더하는 방식으로, 최초 시작은 영점(0)으로 시작하여, 처음으로 1이 나올때까지 0을 point doubling 하여 준다. 0을 point doubling 하는 경우 0에 0을 곱하는 것으로 0이 되며, 1이 나타나면 point add 를 수행하여 최초  $P$ 가  $Q$ 에 더해진다( $0 + P$ ). 이후 다음 루프로 넘어가며 기존의 점  $Q$ 에 대하여 point doubling 을 수행하며,  $2Q$ 가되고, 만약  $n$ 의  $i+1$ 번째 비트가 1이라면 point add 를 수행하고 0이면 수행하지 않고 다음 루프로 넘어가게 된다. 이런식으로  $n$ 에 대하여 MSB로 부터 LSB(Least Significant Bit)까지 스캐닝하며 스칼라 곱을 수행한다.

Sliding-window method(알고리즘-2)는 스칼라 곱의 계수( $n$ )를 window size( $w$ )단 위로 잘라 연산 하는 방식으로, point doubling에 비해 느린 point addition연산을 몰아

서 하는 방식이다. window size 안에서 나타나는 비트값의 합을  $t$ 에 저장하고, 현재의 점( $Q$ )에 대하여  $w$ 만큼 doubling을 수행하여 앞으로 당긴 뒤, 기저( $P$ )를  $t$ 배 한 값( $tP$ )을 더하여 준다. 이후 다시  $n$ 을 window size만큼 잘라 위를 반복 수행한다.

---

**Algorithm 2:** Sliding-window method

---

```

Require:  $P, n$ 
1:  $Q = 0$ 
2: for  $i$  from  $m \dots 0$  do
3:   if  $n_i == 0$  then
4:      $Q = point_{double}(Q)$ 
5:   else
6:      $t = extra\ additional\ bits\ (j)\ from\ n\ (up\ to\ w - 1)$ 
7:      $i = i - j$ 
8:     if  $j < w$  then
9:        $double_{and\_add}(Q, t)$ 
10:    return  $Q$ 
11:   else
12:     while  $(w = w - 1) > 0$  do
13:        $Q = point\ double(Q)$ 
14:     end while
15:      $Q = point_{add}(Q, tP)$ 
16:   end if
17: end if
18: end for
19: return  $Q$ 

```

---

wNAF method(알고리즘-3)는 Sliding-window method와 비슷하나, 0이 연속으로 나오는 경우 Point Addition 연산횟수를 줄여 성능을 향상시키는 방식이다. wNAF는 포인트 연산 전 LSB가 1일 때  $w$ 크기만큼  $d_j$ 에 예 저장하는데( $d = \{d_{i-1}, d_{i-2}, \dots, d_0\}$ ),  $w$ 크기 단위로 자르는 sliding window에 비해 0이 연속으로 많이 나오는 경우 sliding-window method에 비하여 더 적은 횟수의 point addition을 할 수 있는 특징을 가진다. 하지만  $d$ 를 미리 계산해야 하는 precomputation 단계가 선행되어야 한다.  $d$ 가 계산되고 난 이후엔 double-and-add와 같이  $d_j$ 의 값이 0인 경우 point doubling만 수행하고, 0이 아닌 경우  $d_j P$ 만큼 point addition을 수행하여 한 번에 더한다. 한번에  $P$ 씩 더하는

기본 double and add 방식에 비하여 한번에 add를 수행하여 연산 속도를 높일 수 있다.

---

**Algorithm 3:** windowed non-adjacent form(wNAF) method

---

```
Require:  $P, d$ 
1:  $i = 0$ 
2: while  $d > 0$  do
3:   if ( $d \text{ mod} 2$ ) == 1 then
4:      $d_i = d \text{ mod} s2^w$ 
5:      $d = d - d_i$ 
6:   else
7:      $d_i = 0$ 
8:   end if
9:    $d = d/2$ 
10:   $i = i + 1$ 
11: end while
12:  $Q = 0$ 
13: for  $j = i - 1 \dots 0$  do
14:    $Q = \text{point double}(Q)$ 
15:   if  $d_j \neq 0$  then
16:      $Q = \text{point add}(Q, d_j P)$ 
17:   end if
18: end for
19: return  $Q$ 
```

---

montgomery ladder(알고리즘 -4)는 부채널공격(side-channel attack)에 강한 특성을 가진 방법으로 매번 point add와 point doubling을 수행하여 비밀키를 추측하기 어렵게 하는 방식이다. 부채널 공격은 소요시간 분석, 전력 모니터링 공격, 차분 오류 분석 등이 있는데, 매번 같은 연산을 수행함으로써 전력변화 및 소요시간을 동일하게 하여 부채널 공격에 강한 내성을 가지게 된다. montgomery ladder는 기존 double-and-add 방식과 같이 매 비트를 확인하나, double and add와 다른 점은 매 루프마다 point doubling과 point addition을 수행한다는 것이다. 매번 point doubling과 point addition 수행시 현재 스칼라( $n$ )에 따라  $R_0$ 에 point doubling을 저장할지,  $R_1$ 에 point point doubling을 저장할지 정해지는데, 0인 경우  $R_0$ 에 point doubling 을 저장하고,  $R_1$ 에 point addition을 저장한다. 1인 경우엔  $R_1$ 에 point doubling을 저장하고,  $R_0$ 에

point addition을 저장한다. 이때, 항상 point addition은 point doubling보다 1번 더 기저( $P$ )를 더한 상태가 된다( $|R_0 - R_1| = P$ ).

---

**Algorithm 4:** montgomery ladder

---

```

Require:  $P, d$ 
1:  $R_1 = 0$ 
2:  $R_2 = P$ 
3: for  $i = m \dots 0$  do
4:   if  $d_i = 0$  then
5:      $R_1 = pointadd(R_0, R_1)$ 
6:      $R_0 = pointdouble(R_0)$ 
7:   else
8:      $R_0 = pointadd(R_0, R_1)$ 
9:      $R_1 = pointdouble(R_1)$ 
10:  end if
11: end for
12: return  $R_0$ 
```

---

#### 2.4.3.2 프라임 필드 연산

타원곡선상의 점은 프라임 필드에 갇혀있어 정의된 연산 수행 시 필드 안의 값이 나와야 한다. 이때 필드의 크기는 소수이기 때문에 모든 0이 아닌 원소에 대하여 환연산이 되며, 이에 덧셈과 곱셈 그리고 인버전(나눗셈)에 대하여 정의한다. 나눗셈은 프라임 필드 상에 정의되어 있지 않기 때문에 역수를 취해주어 계산하며, 이때 확장 유clidean 알고리즘을 이용하여 역수를 구한다.

타원곡선 암호화를 위해 필요한 연산은 필드 덧셈(field addition), 필드 뺄셈(field subtraction , differnce), 필드 곱셈(field mutiplication), 필드 제곱(field squareing), 필드 역원(field inversion)으로 총 5가지 연산이 정의되어야 한다.

필드 덧셈은 기존의 덧셈 방식과 같으나, 프라임 필드 안에 있어야 하므로 프라임 ( $P$ )보다 커지는 경우  $P$ 를 빼줘 필드 안으로 들어오게 한다. 또한, 뺄셈은 덧셈에 대한 역원을 더하는 것과 같다.

필드 곱셈(알고리즘-6), 필드 스퀘어링 연산 또한 필드 덧셈과 같이 일반적인 곱셈

---

**Algorithm 5:** Prime Field Addition

---

**Require:**  $out, in, P$   
**Ensure:**  $, out$  : 입력1 및 결과 저장 버퍼,  $in$  : 입력2 버퍼,  $P$  : 프라임  
1:  $c = 0$   
2:  $c = out + in$   
3: **if**  $c \geq p$  **then**  
4:    $c = c - P$   
5: **end if**  
6:  $out = c$   
7: **return**  $out$

---

방식과 같으며, 이후 리덕션 연산을 통하여 프라임 필드 안으로 들어오게 한다. 필드 인버스 연산은 페르마 소정리를 이용한 방식과 확장 유클리드를 이용하는 방식이 있다. 페르마 소정리는 프라임 필드 내에서  $a$ 가 정수일 때,  $a^P = a(modP)$ 인 특성을 이용하는 것으로,  $a^{P-1} = 1(modP)$ 이며,  $a^{P-2} = a^{-1}(modP)$  됨으로,  $a$ 를  $P - 2$  제곱하여 역 원을 구한다. 확장 유클리드 방식은  $ns + at = 1$ 일 때 양변에  $(modn)$ 을 취해줌으로써,  $at = 1(modn)$ 으로  $t$ 는  $a$ 의 역원이 되는 특성을 이용한 계산 방식이다(알고리즘-7).

---

**Algorithm 6:** Field Multiplication

---

**Require:**  $a, b$   
1:  $r_0 = 0, r_1 = 0, r_2 = 0$   
2: **for**  $k$  from  $0 \dots 2(t-1)$  **do**  
3:   **for each**  $i, j (i+j = k, 0 \leq i, j < t)$  **do**  
4:      $uv = a_i * b_j$   
5:      $r_0 = add(r_0, v)$   
6:      $r_1 = add with carry(r_1, u)$   
7:      $r_2 = add with carry(r_2, 0)$   
8:   **end for**  
9:    $c_k = r_0$   
10:    $r_0 = r_1$   
11:    $r_1 = r_2$   
12:    $r_2 = 0$   
13: **end for**  
14:  $c_{2t-1} = r_0$   
15: **return**  $c$

---

---

**Algorithm 7:** exntended euclid

---

**Require:**  $a, n$ 

```
1:  $t = 0, newt = 1, r = n, newr = a$ 
2: while  $newr \neq 0$  do
3:    $quotient = r / newr$ 
4:    $newt = t - quotient * newt$ 
5:    $t = newt$ 
6:    $newr = r - quotient * newr$ 
7:    $r = newr$ 
8: end while
9: if  $r > 1$  then
10:  return -1
11: end if
12: if  $t < 0$  then
13:   $t = t + n$ 
14: end if
15: return  $t$ 
```

---

필드 곱셈 연산과 필드 스퀘어링 연산은 리덕션 연산을 통하여 프라임 필드 안으로 들어오게 하는 방법이 필요하며, 이를 위한 fast reduction, barret reduction 과 같은 다양한 리덕션 기법이 있다.

## 2.5 Curve25519

지난 2013년 NIST DUAL\_EC\_DRBG에서 백도어가 발견됨으로써 NIST에서 제시하는 타원곡선은 사실상 사용이 배제되기 시작했다 J. Berstein과 Tanja Lange가 NIST 커브에 대하여 안전하지 않음을 발견하여 기존 타원곡선 암호를 대체할 곡선을 찾아야 했다. 이에 떠오르는 차세대 타원곡선으론 2005년 Daniel J. Bernstein이 제시한 Curve 25519가 유력한 후보로 올라와 있으며 이미 GNUTLS와 OpenSSL등 많은 암호 라이브러리에서 이미 지원하고 있다.

### 2.5.1 Curve25519의 특징

Curve25519는  $y^2 = x^3 + 486662x^2 + x$ 의 몽고메리 타원곡선을 이용하여 기존  $y^2 = x^3 - 3x + a_6$ 보다  $(486662 - 2)/4$ 만큼 작은 크기를 가진다. 프라임 필드로 사용되는 소수는  $2^{255} - 19$ 이며,  $F_p$ 는  $Z/p = Z/2^{255} - 19$ 이며, 기저( $P$ )로  $x = 9$ 를 사용한다.

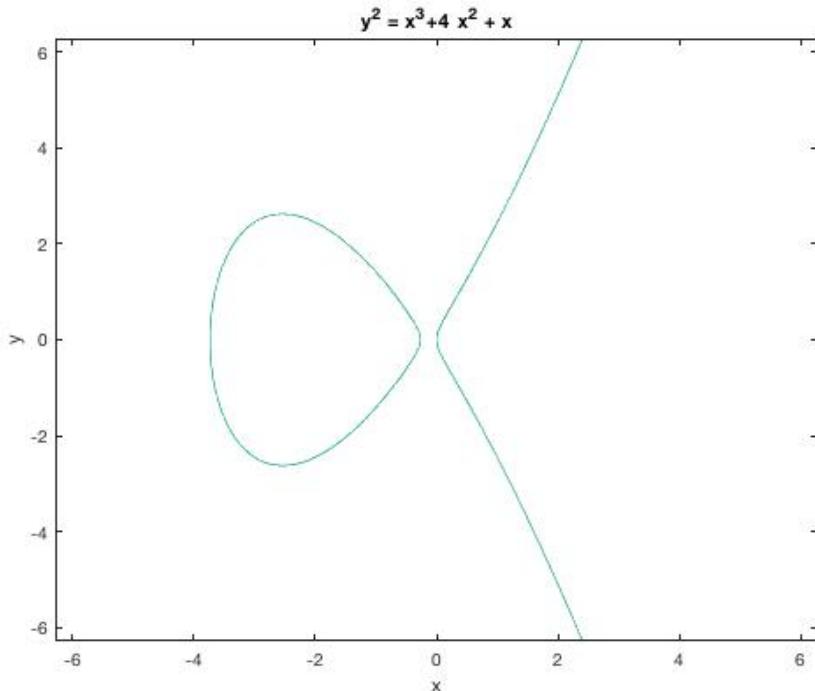


Figure 9:  $y^2 = x^3 + 4x^2 + x$  몽고메리 타원곡선

특히 Curve25519는 스칼라 곱 연산 시  $x$ 와  $z$  값만을 이용하는 특징이 있는데,  $y$  값 까지 사용하는 NIST curve에 비하여 적은 메모리를 사용하여 메모리 최적화에 유리하다.

Curve25519는 부채널공격에 강한 내성을 위해 스칼라 곱 연산시 몽고메리 사다리 (montgomery ladder-10)를 추천하고 있으며, 몽고메리 사다리를 연산하기 위해  $(x, z) =$

$Q$ 와  $(x', z') = Q'$ 이 필요하다. 몽고메리 사다리를 통과하게 되면  $(x_2, z_2) = x_2/z_2 = 2Q$  와  $(x_3, z_3) = x_3/z_3 = 2Q + Q$ 가 나온다.

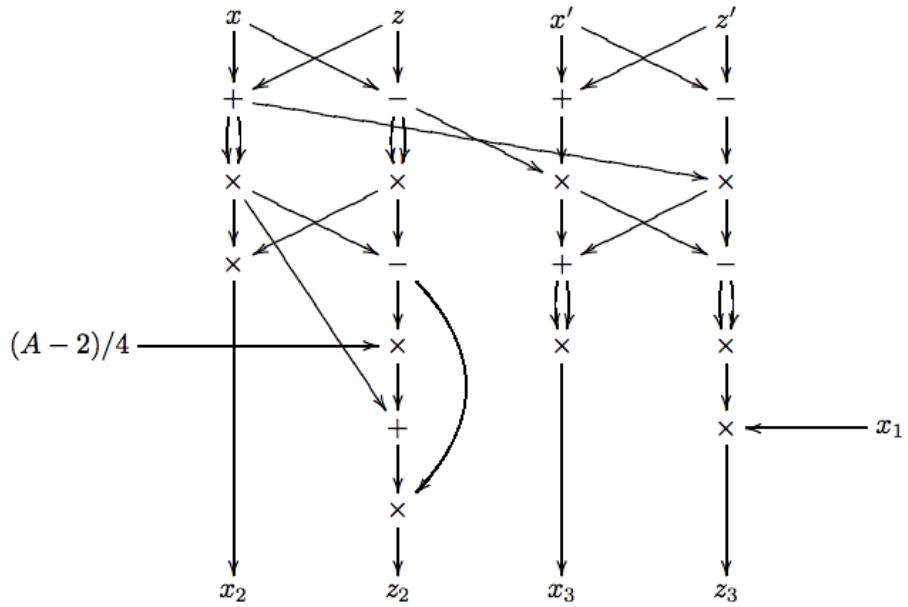


Figure 10: Curve25519상 몽고메리 사다리

## 2.6 ECDH

ECDH(Elliptic Curve Diffie-Hellman)은 키 교환 알고리즘으로 두 사용자간 공개 키를 교환하여 서로 같은 대칭 키를 만드는 방식이다. 그림-11는 Curve25519를 사용하는 ECDH 키교환 방식을 나타낸다. 두 사용자의 비밀키 (Alice :  $a$ , Bob :  $b$ )를 기저 (Base Point)의 값인 9와 스칼라 곱을 하여 각각 공개키 (Alice :  $\text{Curve25519}(a, 9)$ , Bob :  $\text{Curve25519}(b, 9)$ )를 생성하고, 이를 서로 교환한다. 이후 Alice는 Bob으로부터 받은 Bob의 공개키( $\text{Curve25519}(b, 9)$ )에 Alice의 비밀키인  $a$ 를 곱하여 대칭키 ( $\text{Curve25519}(a, \text{Curve25519}(b, 9))$ )를 생성하고, Bob은 Alice로부터 받은 공개키( $\text{Curve25519}(a, 9)$ )에 Bob의 비밀키인  $b$ 를 곱하여 대칭키( $\text{Curve25519}(b, \text{Curve25519}(a, 9))$ )를 생성한다.

이때 생성된 대칭키는 서로 같으므로 AES, SEED, ARIA, HIGHT와 같은 대칭키 암호화 방식에 사용할 수 있다.

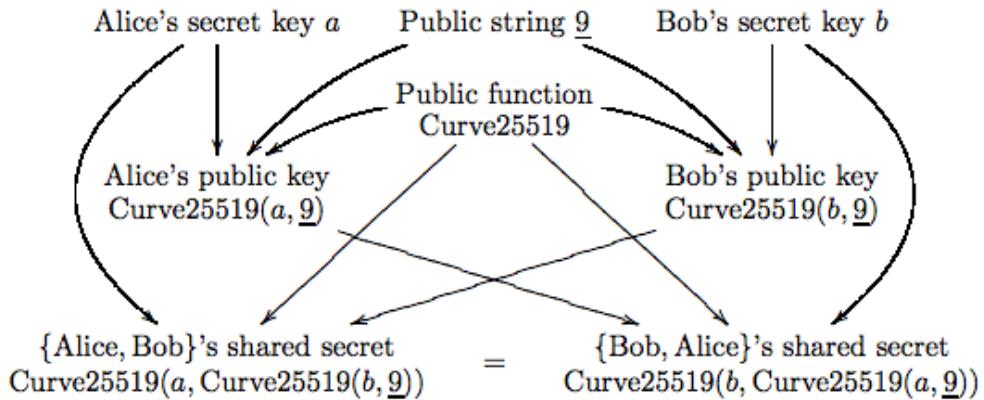


Figure 11: Curve25519를 사용하는 ECDH 키교환

---

**Algorithm 8:** ECDH 검증 알고리즘

---

**Require:**  $P, Q, B$   
**Ensure:** ,  $P$  : Key1,  $Q$  : Key2,  $B$ :Base Point  
 1:  $e1k = \text{Curve25519}(Q, B)$   
 2:  $e2e1k = \text{Curve25519}(P, e1k)$   
 3:  $e2k = \text{Curve25519}(P, B)$   
 4:  $e1e2k = \text{Curve25519}(Q, e1k)$   
 5: **if**  $e1e2k \neq e2e1k$  **then**  
 6:     **break**  
 7: **end if**  
 8: **return** 1

---

### 3 GPGPU 성능 향상 기법

GPGPU를 활용하여 성능을 향상시키는 방법은 크게 메모리 사용량과 코어 개수를 고려한 다중쓰레딩, 루프 언롤링, 코드 직렬화, 다중 다중 병렬로 크게 4가지 기법이 있다. 이 이외에도 코드 최적화와 같이 컴파일러에서 제공하는 기능과 파이프라이닝을 고려한 연산 배치 등의 방법이 있다.

#### 3.1 다중 쓰레딩

GPGPU는 CPU에 비하여 많은 코어를 가지고 있다. CPU는 보통 1개에서 8개 정도의 복잡한 연산을 수행할 수 있는 강력한 성능을 가진 코어로 구성되어 있는 반면, GPU는 수백에서 수천 개의 작은 코어로 구성되어 있다. 각각의 성능은 좋지 않지만, 그래픽처리 및 산술연산에 특화된 산술논리 연산장치(Arithmetic and Logic Unit, ALU)를 가지고 있어 대량의 데이터에 대하여 간단한 연산을 빠르게 수행할 수 있다. 이러한 특징으로 GPGPU는 한 번에 수백, 수천 개의 코어를 동작시켜 CPU에 비하여 간단한 동작에 대한 연산에 대하여 성능을 향상시킬수 있다. 또한 GPU는 빠른 입출력속도를 가지는 여러 개총의 메모리로 구성되어있어 다중 쓰레딩처리를 할 경우 쓰레드별 상호작용 및 고속처리가 가능하다.      다중 쓰레드처리 시 성능은 사용하는 쓰레드에 비례하여 선형적으로 증가하며, 디바이스가 수행할 수 있는 쓰레드의 최대치에 도달하면 성능이 떨어지게 된다.

#### 3.2 루프 언롤링(Loop Unrolling)

루프 언롤링은 반복문을 직렬화하는 것으로 for 루프 혹은 while 루프 동작 시 불필요한 분기문을 제거하여 속도를 향상시키는 방법이다. for루프와 while루프는 매 루프마다 조건문을 확인하여 참인 경우 루프 내 코드를 수행하고, 거짓인 경우 루프 밖

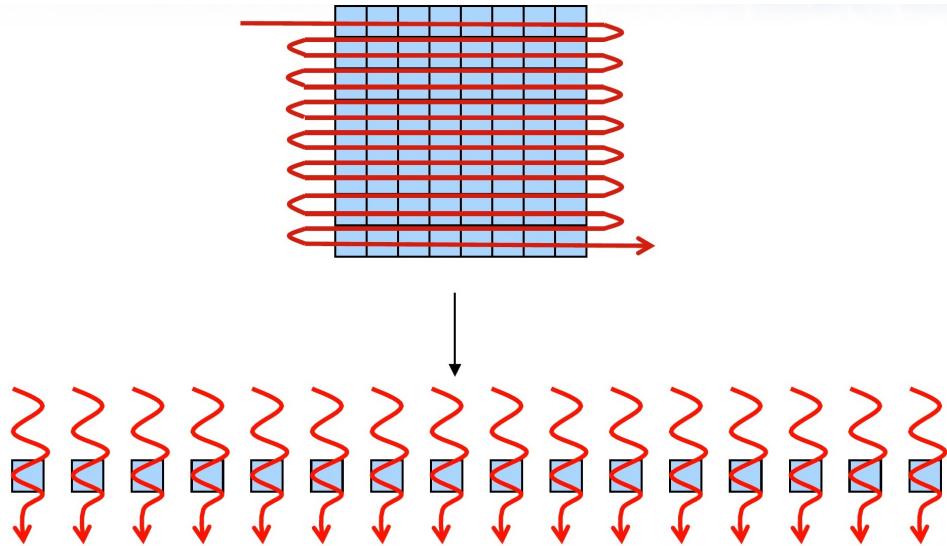


Figure 12: 싱글 쓰레드와 멀티 쓰레딩

으로 탈출하는 동작을 하며, 이때 조건문이 복잡할수록 성능이 하락하게 된다. 루프가  
도는 횟수가 고정이거나 횟수가 적은 경우 루프 언롤링을 하여 성능을 향상시킬 수 있  
는데, GPGPU에서도 마찬가지로 루프언롤링을 적용하면 성능이 향상된다. 다음은  
루프언롤링의 예시이다.

```
void rolling( int* a, int* b){
    for ( int i = 0 ; i < 4 ; i++)
        a[ i ] = a[ i ] + b[ i ];
    return ;
}
```

```
void unrolling( int *a, int* b){
    a[ 0 ] = a[ 0 ] + b[ 0 ];
    a[ 1 ] = a[ 1 ] + b[ 1 ];
```

```

    a[2] = a[2] + b[2];
    a[3] = a[3] + b[3];
    return;
}

```

반복문을 제거하고 같은 동작을 명시적으로 모두 작성하여 루프 언롤링을 구현할 수 있으나, 이때 반복문의 횟수가 고정되어 있어야 가능하다. 반대로 루프의 횟수가 동적 이거나 알 수 없는 경우엔 루프 언롤링을 적용할 수 없다.

### 3.3 코드 직렬화

코드 직렬화는 함수호출횟수를 줄여 불필요한 연산을 줄이는 방법이다. 같은 동작을 하는 코드의 경우 함수로 만들어 코드의 중복을 줄여 코드사이즈를 줄여 메모리사용량을 줄일 수 있지만, 메모리의 크기가 큰 GPGPU의 경우 코드사이즈는 크게 문제가 되지 않는다. 또한, 함수가 호출되면 함수 인자 전달을 위한 값 복사 및 분기 문이 발생하게 되어 파이프라인이 깨지게 되어 성능 하락의 원인이 된다. 이 때문에 같은 동작을 하는 코드일지라도 함수로 만들지 않고 작성하여 불필요한 분기 문과 불필요한 메모리 입출력을 줄여 성능을 향상 시킬수 있다. 다음은 코드직렬화의 예시이다.

```

void add( int *a, int* b){
    *a = *a + *b;
}

// undirected function
void undirectedF( int* a, int* b){
    for ( int i = 0 ; i < 4 ; i++)

```

```

        add( a + i , b + i );

    return ;
}

// directed function
void directedF( int *a , int* b){

    a[0] = a[0] + b[0];
    a[1] = a[1] + b[1];
    a[2] = a[2] + b[2];
    a[3] = a[3] + b[3];

    return ;
}

```

위 코드는 루프언롤링과 코드 직렬화가 동시에 적용된 모습이다. 루프를 해제하면서 add 함수가 4번 호출되게 되고 이러한 4번의 호출에 대하여 코드 직렬화를 적용하여 함수의 기능을 직렬화 함수에 적용하여 함수호출 횟수를 줄였다.

### 3.4 다중 병렬화

다중병렬처리는 GPGPU에서 병렬처리 중 한번 더 병렬 처리를 하는 것이다. 그림-13은 다중 병렬처리를 나타낸다. 다중병렬처리는 이미 시작된 커널 내에서 다시 병렬처리하는 것이 특징인데 하나의 쓰레드에서만 분기하여 병렬처리를 하거나 여러 개의 쓰레드에서 또다시 병렬처리 하는것이다. 이때 동기화 문제가 발생하게 되는데 부포 쓰레드에서 자식 쓰레드를 기다려주거나 기다려주지 않고 종료하는 이슈가 발생 한다. 여러개의 자식쓰레드가 생기는경우 최대 부모쓰레드의 개수만큼 생겨  $N^2$ 개의

쓰레드가 생성 될수 있으며, 이를 동기화 하는경우  $N^2 * time$  만큼의 시간이 추가로 들수 있어 사용에 주의가 필요하다.

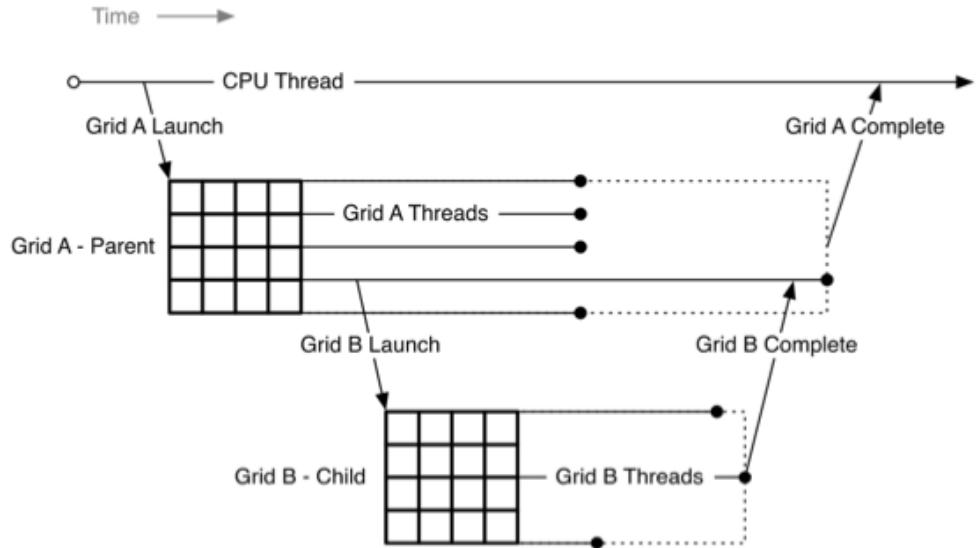


Figure 13: 다중 병렬화

CUDA의 경우 다중병렬처리는 compute35기종 이상에서부터 지원하며, Generate Relocation Device Code를 추가로 생성해줘야 다중병렬처리가 가능하다. 다중 병렬처리를 지원하게되면 디바이스에서 다시 커널을 호출이 가능하게된다. 이때 파라미터로 받는 데이터는 모두 디바이스에서 할당되어 접근가능한 상태이어야 한다.

### 3.5 메모리 정렬

GPGPU는 메모리를 뱅크(bank)단위로 나누어 사용하는데, 이는 64바이트 단위로 나누어져있다. 쓰레드에서 메모리 접근시 16개 쓰레드에서 순차적으로 4byte씩 접근하는 경우 64바이트에 접근함으로 한번의 엑세스로 모두 읽어올수 있다. 하지만 64바이트가 아니거나 동시에 같은메모리 위치에 접근하는경우 2번 엑세스가 일어나 성능은 절반으로 떨어지게 된다. 또한 16개의 쓰레드가 하나의 뱅크에 접근하는경우 16

번의 엑세스가 발생하게되어 성능은 1/16으로 떨어지게 된다. 이 때문에 배열 생성시 1단계씩 어긋나게 메모리를 추가로 할당하여 충돌을 방지할수 있다.

### 3.6 연산 정렬

GPGPU의 명령어는 하나의 메모리로부터 읽어 같은 라인에 있는 ALU에 모두 적용되기 때문에 같은 라인에 있는 쓰레드들이 같은 연산을 수행할때 최대의 효율이 나타난다. 그림-14과 같이 시간 진행대 따라 같은 연산이 수행되는 경우 쓰레드의 대기 시간없이 수행되게 되나, 알고리즘-9와 같이 수행되면 그림-15 처럼 쓰레드별로 다른 연산을 수행해 유휴쓰레드가 많이 생겨 속도가 저하되게 된다. 따라서 프로그램 작성시 분기문(if, switch)와 같은 명령어로 분기를 만드는것은 지양해야 한다.

사이클	시간 진행					
	0	1	2	3	4	5
0번 쓰레드	+	-	*	+	*	-
1번 쓰레드	+	-	*	+	*	-
2번 쓰레드	+	-	*	+	*	-
3번 쓰레드	+	-	*	+	*	-
...						
30번 쓰레드	+	-	*	+	*	-
31번 쓰레드	+	-	*	+	*	-

Figure 14: 연산 정렬

---

**Algorithm 9:** 유~~한~~ 쓰레드가 발생하는 처리

---

**Ensure:** ,  $tid$  : 쓰레드 아이디

- 1:  $a[tid] = tid$
  - 2: **if**  $tid \% 2 == 0$  **then**
  - 3:    $a +=$
  - 4: **else**
  - 5:    $a -=$
  - 6: **end if**
  - 7:  $a = a * a$
- 

시간 진행 

사이클	0	1	2
0번 쓰레드	+	대기	*
1번 쓰레드	대기	-	*
2번 쓰레드	+	대기	*
3번 쓰레드	대기	-	*
30번 쓰레드	+	대기	*
31번 쓰레드	대기	-	*

Figure 15: 연산 정렬

## 4 GPGPU상 Curve25519 구현 및 성능 향상 방법

해당 장에서는 curve25519의 구현방법과 필드 곱셈 연산에 대하여 10개의 쓰레드로 병렬처리하여 성능 향상 정도를 측정하고, 스칼라 곱에 대한 성능 향상 방안에 대해 설명한다. 그리고 ECDH을 구현하여 성능 향상방법을 적용한 결과를 비교하여 성능 향상에 어느정도 도움이되는지 측정한 결과를 분석한다. 적용된 성능향상 기법은 다중 쓰레딩기법, 루프 언롤링, 및 코드 직렬화로 3가지 성능 향상 기법을 각각 적용 하였다. 각실험은 각각의 케이스에 대하여 5회씩 수행된 평균값이며 소수점자리는 반올림 하였다.

### 4.1 타겟 GPU및 구현 환경

GTX 1050Ti는 6개의 쿠다코어(cuda core)를 가지고 있으며 각각의 쿠다코어는 128개의 SM이 들어있다., 4GB의 GDDR5버퍼와 7Gbps의 메모리 속도를내는 GPU로 메인보드의 PCI슬롯에 장착되어 사용된다. 구현환경은 Visual Studio 2015 Community 버전에서 CUDA 8.0라이브러리를 사용하여 구현되었다. 비교대상 CPU는 Intel i76700 3.4GHz 이며, 16GB 램을 가지고 있다. ECDH연산에 대한 검증을 하기위해 한 쌍의 키에 대하여 두번의 스칼라 곱연산과 순서를 바꾼 두번의 스칼라 곱 연산으로 ECDH연산을 두번 수행하여 총 4번의 스칼라 곱연산을 수행하여 두 결과값이 맞는지 확인한다.

### 4.2 curve25519 구현

curve25519를 구현하기 위해 스칼라 곱으로 몽고메리 사다리 방식을 사용하였고, 역원을 구하기 위해서 페르마 소정리를 이용하였다. 포인트를 저장하기 위해 64bit(uint64\_t) 변수 10개를 가지는 배열을 사용 하였으며 각 변수는 26bit단위로 사용 하였다( $2^{26} * 10 =$

260bit), 필드 곱 연산시 배열은 19개를 사용하며, 리덕션 이후 10개의 크기를 사용하는 배열로 전환한다. curve25519를 구현한 전체 알고리즘은 알고리즘-10와 같다.

---

**Algorithm 10:** curve25519

---

**Ensure:**  $p$  : 생성된 공개키,  $q$  : 비밀키,  $b$  : 기저,  $x$  :  $x$  좌표,  $z$  :  $z$  좌표,  $z'$  :  $z$ 의 곱셈에 대한 역원,  $E$  : 비밀키 배열

- 1:  $E = q$
- 2:  $bp = expand(b)$
- 3:  $(x, z) = scalar_{multiplication}(E, bp)$
- 4:  $z' = inverse(z)$
- 5:  $p = x * z'$
- 6: **return**  $p$

---

기저( $b$ )를 스칼라 곱 하기위해 비밀키  $q$ 를 비밀키 배열인  $E$ 로 복사한다. 이후 문자열로 받은  $b$ (base point)를  $bp$ 배열로 옮기는 작업을 수행한다(expand).  $bp$ 와  $E$ 를 스칼라 곱을 수행하여  $x$ 값과  $z$ 값을 구하며, 마지막으로 공개키  $p$ 를 계산하기위해  $x/z = x * z^{-1} = x * inverse(z)$ 를 계산한다. 스칼라 곱셈은 몽고메리 사다리방식(알고리즘-11)을 이용하였으며, 이때 사용한 곱셈식은 알고리즘-12과 같다. 몽고메리 사다리 방식은 그림-10과 같으며 point addition을 먼저 수행한뒤, point doubling을 마지막에 계산하였다(그림-16).

#### 4.2.1 필드 곱셈 병렬화

스칼라 곱셈 내부연산은 프라임 필드( $F(P) = \{0...2^{255} - 20\}$ )에 정의된 연산으로, 이때 10개로 나눠진 배열을 사용하여 덧셈, 곱셈에 대하여 병렬 처리가하다. 기존의 필드 곱셈 방법은 그림-17와 같이 1개의 코어가 100번의 곱셈 연산을 한뒤, 81번의 덧셈 연산을 해 줘야 한다. Curve25519는 각각의 배열에 대하여 26bit만큼 사용하여 정수를 표현하며,  $2^{255}x^{10} - 19$ 안에 있어야한다. 이러한 필드 곱셈 구현시 짹수번째 배열은

---

**Algorithm 11:** curve25519 스칼라곱(몽고메리 사다리 사용)

---

**Require:**  $n, q$   
**Ensure:**  $n$ : 스칼라 곱셈 계수(비밀키),  $q$ : 기저(baise point)  
**Ensure:**  $x_2$  : point doubling 결과 x좌표,  $z_2$  : point doubling 결과 z좌표,  
**Ensure:**  $x_3$ : point addition 결과 x좌표,  $z_3$ : point addition 결과 z좌표

- 1:
- 2: **for**  $i = 0 \dots 255$  **do**
- 3:      $bit = n_i$
- 4:     **if**  $bit = 1$  **then**
- 5:          $swap(x, x'), swap(z, z')$
- 6:     **end if**
- 7:      $(x_2, z_2, x_3, z_3) = montgomery\_point\_double\_and\_add(x, z, x', z', q)$
- 8:     **if**  $bit = 1$  **then**
- 9:          $swap(x_2, x_3), swap(z_2, z_3)$
- 10:     **end if**
- 11:      $swap(x, x_2), swap(z, z_2), swap(x', x_3), swap(z', z_3)$
- 12: **end for**
- 13:  $p = (x_2, z_2)$
- 14: **return**  $p$

---

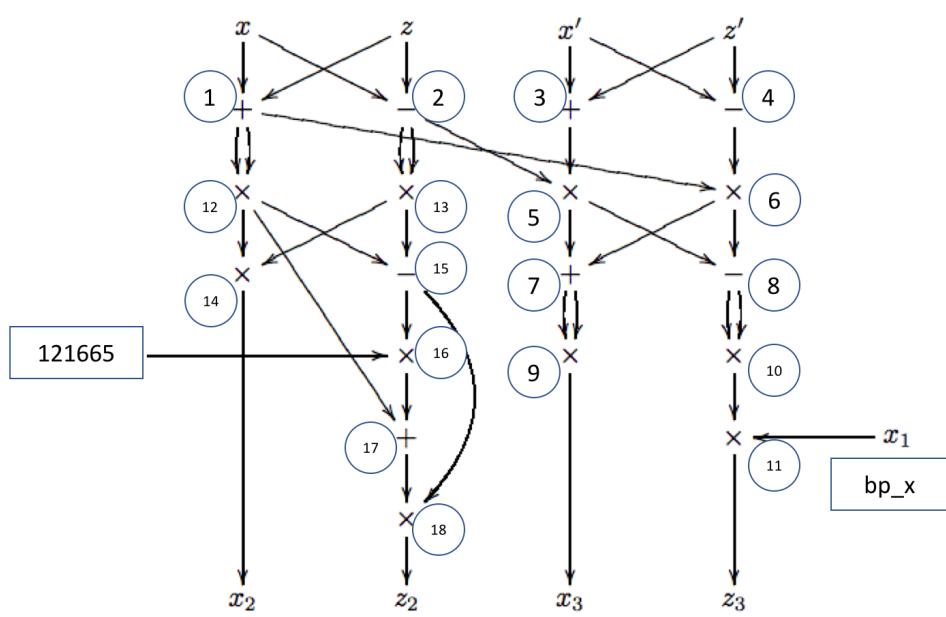


Figure 16: 몽고메리 double and add 연산 순서

---

**Algorithm 12:** curve25519 몽고메리 double-and-add

---

**Require:**  $x_2, z_2, x_3, z_3, x, z, x', z', q$   
**Ensure:**  $x_2$  : output  $2Q$ 의  $x$ ,  $z_2$  : output  $2Q$ 의  $z$   
**Ensure:**  $x_3$  : output  $Q+Q'$ 의  $x$ ,  $z_3$  : output  $Q+Q'$ 의  $z$   
**Ensure:**  $x$  : input  $Q$ 의  $x$ ,  $z$  : input  $Q$ 의  $z$   
**Ensure:**  $x'$  : input  $Q'$ 의  $x$ ,  $z'$  : input  $Q'$ 의  $z$ ,  $q$  : 기저 포인트

- 1:  $origin\_x = x$
- 2:  $x = x + z$
- 3:  $z = z - origin\_x$
- 4:  $origin\_x' = x' + z'$
- 5:  $x' = x' + z'$
- 6:  $z' = origin'_x + z'$
- 7:  $x'' = x' * z$
- 8:  $z'' = x * z'$
- 9:  $origin'_x = x''$
- 10:  $x'' = x'' + z''$
- 11:  $z'' = z'' - origin'_x$
- 12:  $x''' = x'' * x''$
- 13:  $z''' = z'' * z''$
- 14:  $z'' = z''' * q.x$
- 15:  $x_3 = x''$
- 16:  $z_3 = z''$
- 17:  $x'' = x * x$
- 18:  $z'' = z * z$
- 19:  $x_2 = x'' * z''$
- 20:  $z'' = z'' - x''$
- 21:  $z''' = z'' * 121665$
- 22:  $z''' = z''' + x''$
- 23:  $z_2 = z'' * z'''$
- 24: **return**  $(x_2, z_2), (x_3, z_3)$

---

26bit, 훌수번째 배열은 26비트 값에서 하위 1비트를 0으로 하여 carry처리 효율을 높이는 방법을 사용하기 위해 2배를 하여 짝수로 만들어 넣어준다. 이러한 곱셈은 10개의 배열을 10개의 쓰레드를 이용하여 동시에 더할 수 있으며, 곱셈은 그림-18와 같이 연산이 가능하다. 10개의 원소를 가진 배열 두개를 곱하는 경우 19크기의 배열이 필요하며,  $x^{10}$ 부터  $x^{18}$ 까지의 값은 리덕션을 통하여 지워야 한다. 이때 curve25519는 간단히  $x^{10}$  배 위치의 가수에 대하여 19를 곱하여 더하는 방식으로 계산할 수 있다 (알고리즘-13).

Figure 17: 필드 곱셈

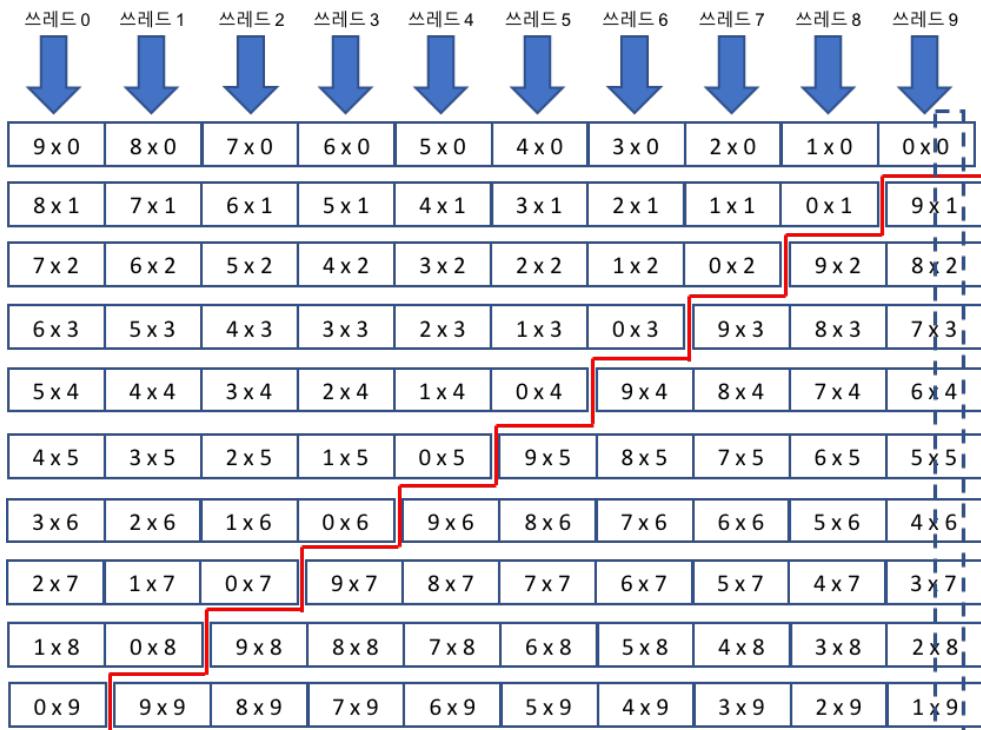


Figure 18: 필드 곱셈 병렬화

필드 곱셈 병렬화시 그림-18와 같이 10개의 쓰레드에서 각각 10번의 곱셈을 수행하고 9번의 덧셈을 수행하여 덧셈이 9번의 추가적인 덧셈이 발생 하였으나, 10개의

쓰레드가 동시에 수행하여 더 빠르게 계산이 가능하다. 표-5는 이를 적용한 필드 곱셈의 수행속도를 비교한 결과이며 300만번 동시 수행부터 GPGPU에서 더 빠른 성능을 나타내었다.

#### 4.2.2 몽고메리 double and add 병렬 방법

몽고메리 사다리에서 사용하기 위한 몽고메리 doubleandaddition 은 4개의 쓰레드를 이용하는 경우 연산 병렬화를 적용 할수 있으며, 기존 18번의 연산을 수행하는 연산에 대하여 연산 병렬화를 통하여 8번만에 계산이 가능하다(그림-19). 연산 병렬화를 할 경우 표-6와 같이 평균 49%의 연산횟수 감소 효과를 볼 수 있다. 특히 9번 수행하는 곱셈 연산을 3번의 사이클만에 수행 할수 있어 속도향상에 큰 영향을 미칠수 있다.

### 4.3 다중쓰레딩

다중쓰레딩에선 한번에 사용하는 cuda코어수를 조절하여 메모리 접근량을 조절하고 동시에 수행되는 코어의 개수를 조절하여 부하를 분산시키는 방법이며 Table-7은 GPGPU에서의 400회의 스칼라 곱 연산수행 횟수이다. 총 연산된 갯수는 쓰레드 개수에 400회의 곱과 같다.

다중쓰레딩방식으로 쓰레드 개수를 10개부터 100개까지 순차적으로 10개씩 증가시켜 400회씩 스칼라 곱 연산을 한 결과는 표-7과 같다. 쓰레드개수가 10인 경우 약 12.6초가 걸렸으며, 10개씩 증가할때마다 약 0.1초씩 시간이 증가하며, 60개부터 속도가 약간 빨라지는 현상을 볼 수 있다.

### 4.4 루프 언롤링

루프로 묶여있는 코드를 직렬화 하는 루프언롤링방식은 코드의 길이가 길어지고 코드 크기가 커지는 문제점이 있지만 큰 메모리 사이즈를 가지는 GPGPU환경에선 크게 문

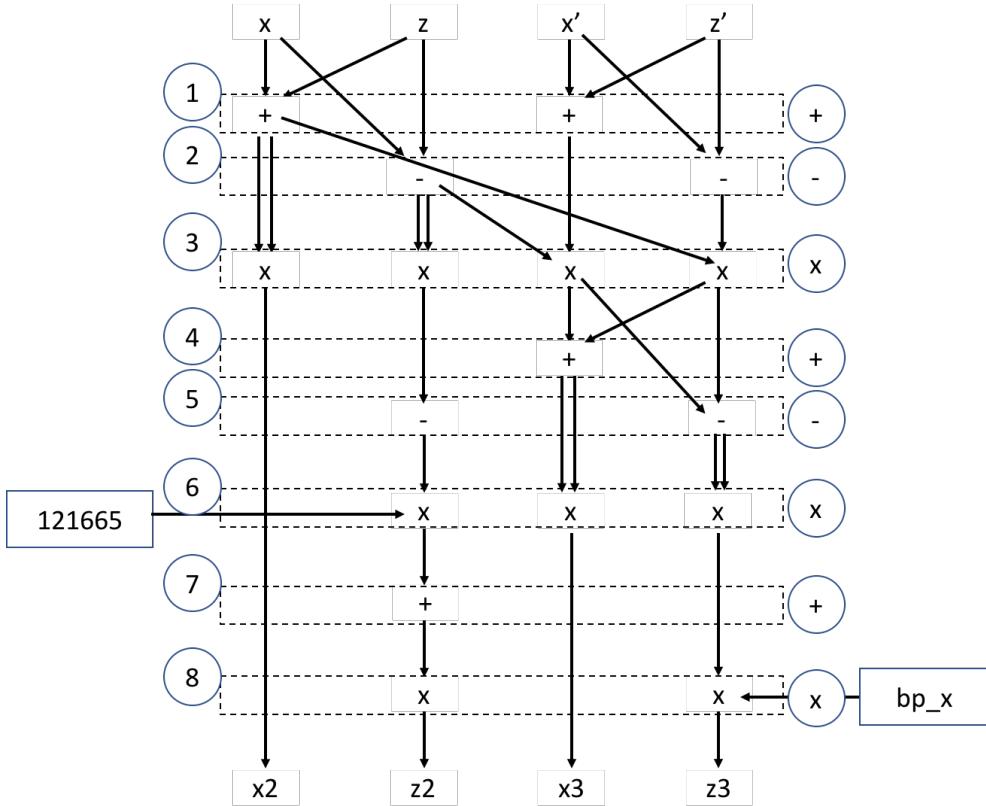


Figure 19: 몽고메리 double and add 연산 병렬화

제가 되진 않는다. Curve25519 연산에서 필드연산중 덧셈, 곱셈, 뺄셈등은 단순 루프를 돌며 연산하기때문에 루프 언롤링방식이 적용될 수 있다. 알고리즘-15은 필드 덧셈에서 unrolling방식이 적용되지 않은 방식이고, 알고리즘-16은 루프 언롤링방식이 적용된 덧셈이다.

본 실험에선 덧셈, 뺄셈, 곱셈, 그리고 리덕션 연산에서 루프언롤링방식을 적용할 수 있었다. 표-9은 루프언롤링방식이 적용되지 않은 결과이고, 표-10은 루프언롤링방식이 적용된 결과이다.

덧셈, 뺄셈, 곱셈과 리덕션에 루프언롤링 방식을 적용한 결과는 표-11와 같으며, 약 1%의 성능 향상이 있었다. 또한, 동시에 수행하는 쓰레드의 개수가 작을수록 성능이

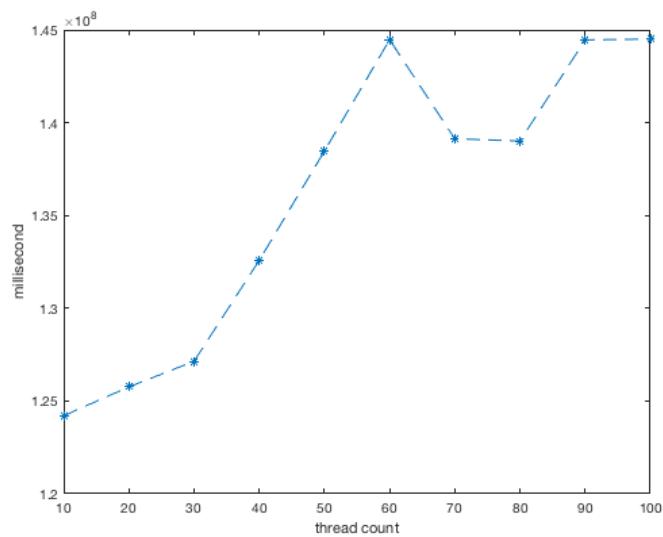


Figure 20: 쓰레드 개수별 수행시간 그래프

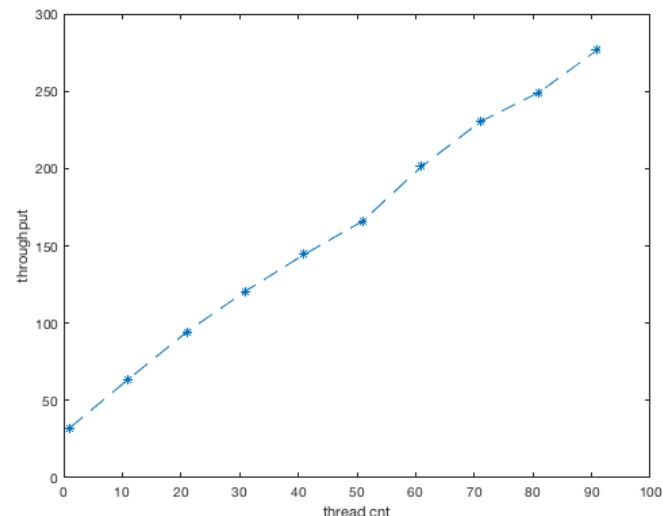


Figure 21: 쓰레드 개수별 처리량 그래프

좋아지는것을 알 수 있었다.

## 4.5 코드 직렬화

코드 직렬화는 함수호출과 같은 분기문 호출 시 현재 상태저장 및 분기문으로 가기 위한 브랜치로 인한 파이프라이닝이 깨지는 문제점을 해결하기 위해 반복되는 동작이나 가독성을 올리기 위한 함수화를 하지 않고 직렬로 작성하는 방법이다. 본 실험에선 함수호출이 잦은 연산에 대하여 코드 직렬화를 적용하였으며, 총 7362번의 함수 호출이 일어나는 환경에서 직렬화를 통해 2392회를 줄여 약 32.5%의 함수 호출을 줄였다.

하지만 코드 직렬화를 하는 경우 표-15에서 보다시피 성능이 떨어지는 것을 확인 할 수 있으며, 이는 함수호출로 인한 속도 감소보다 코드 직렬화로 인한 코드 길이 증가로 최적화가 되지 않아 발생하는 것으로 보인다.

---

**Algorithm 13:** curve25519 필드 곱셈 병렬화 방법

---

**Require:**  $output, in2, in$

**Ensure:**  $tid$  : 블록 전체 쓰레드 id,  $t$  : 블록 내 쓰레드 id

```
1: while  $i = 0 \dots t$  do
2:    $out[tid] += in2[i] * in[t - i]$ 
3:    $i += 1$ 
4: end while
5: while  $i < 10$  do
6:    $out[tid + 10] += in2[i] * in[10 - i + t]$ 
7:    $i += 1$ 
8: end while
9: syncthreads()
10: if  $t < 5$  then
11:    $i = 0$ 
12:   while  $i = 0 \dots t$  do
13:      $out[tid] += in2[i * 2 + 1] * in[(t - i) * 2 - 1]$ 
14:      $i += 1$ 
15:   end while
16:   while  $i < 10$  do
17:      $out[tid + 10] += in2[i * 2 + 1] * in[9 - (i - t) * 2]$ 
18:      $i += 1$ 
19:   end while
20: end if
21: syncthreads()
22: if  $t < 9$  then
23:    $out[tid] += out[tid + 10] * 19$ 
24: end if
25: syncthreads()
26: if  $t == 0$  then
27:    $out[tid + 10] = 0$ 
28:   while  $i = \{0, 2, 4, 6, 8\}$  do
29:      $over = div\_2\_26(out[tid + 0])$ 
30:      $out[tid + i] -= over \ll 26$ 
31:      $out[tid + i + 1] += over$ 
32:      $over = div\_2\_25(out[tid + 1])$ 
33:      $out[tid + i + 1] -= over \ll 25$ 
34:      $out[tid + i + 2] += over$ 
35:   end while
36: end if
37: syncthreads()
38: return  $out$ 
```

---

Table 5: 필드 곱셈 병렬화 속도 비교

횟수(만번)	CPU(ms)	GPGPU(ms)	감소 비율(%)
256	565	656	-16.1
280	624	654	4.8
300	676	649	3.9
330	727	667	8.3
384	880	701	20.3
512	1154	958	17.0
1024	2253	1860	17.4

Table 6: 몽고메리 double and add 병렬화 연산 횟수 비교

연산	직렬	병렬	감소 비율(%)
+	3	2	33
-	4	2	50
*	9	3	66

---

**Algorithm 14:** Multi Threading 알고리즘

---

**Require:**  $Q, B, TC$   
**Ensure:**  $Q$  : Key,  $B$ :Base Point,  $TC$ : Thread Count,  $OUT$ : Output  
1:  $CudaMemcpy(dev1, Q, TC)$   
2:  $CudaMemcpy(dev2, B, TC)$   
3:  $cudaCurve25519 <<< 1, TC >>>$   
4:  $CudaMemcpy(OUT, dev3, TC)$   
5: **return**  $OUT$

---



---

**Algorithm 15:** no-Unrolling fsum

---

**Require:**  $out, in$   
**Ensure:**  $, out$  : 입력1 및 결과 저장 버퍼,  $in$  : 입력2 버퍼  
1: **for**  $i \leftarrow 0..10$  **do**  
2:    $out[i] = out[i] + in[i]$   
3: **end for**  
4: **return** 0

---

Table 7: GTX 1050Ti에서 쓰레드개수별 400회 Curve25519 수행 시간

thread cnt	400회 연산(ms)	400회 연산(s)
10	124204311	124
20	125758874	125
30	127138336	127
40	132577687	132
50	138467993	138
60	144470402	144
70	139135872	139
80	139009875	139
90	144462702	144
100	144506119	144

Table 8: 쓰레드개수별 400회 Curve25519 수행 시간에 대한 처리량

쓰레드 개수	400회 연산(s)	스칼라 곱 횟수	처리량(scalar Mul/s)
10	124	4000	32.2
20	125	8000	63.6
30	127	12000	94.4
40	132	16000	120.7
50	138	20000	144.4
60	144	24000	166.1
70	139	28000	201.2
80	139	32000	230.2
90	144	36000	249.2
100	144	40000	276.8

---

**Algorithm 16:** Unrolling fsum

---

**Require:**  $out, in$

**Ensure:** ,  $out$  : 입력1 및 결과 저장 버퍼,  $in$  : 입력2 버퍼

- 1:  $out[0] = out[0] + in[0]$
  - 2:  $out[1] = out[1] + in[1]$
  - 3:  $out[2] = out[2] + in[2]$
  - 4:  $out[3] = out[3] + in[3]$
  - 5:  $out[4] = out[4] + in[4]$
  - 6:  $out[5] = out[5] + in[5]$
  - 7:  $out[6] = out[6] + in[6]$
  - 8:  $out[7] = out[7] + in[7]$
  - 9:  $out[8] = out[8] + in[8]$
  - 10:  $out[9] = out[9] + in[9]$
  - 11: **return** 0
- 

Table 9: Unrolling 미적용 수행 시간

쓰레드 개수	Unrolling 미적용(ms)
50	13786223
75	13866593
100	14457071

Table 10: Unrolling 적용 수행 시간

쓰레드 개수	Unrolling 적용(ms)
50	13613866
75	13738441
100	14321312

Table 11: Unrolling 적용, 미적용 간 성능 향상

쓰레드 개수	시간 차(ms)	성능향상(%)
50	172357	1.25
75	128152	0.92
100	135759	0.93

Table 12: 코드 직렬화 비율

함수 호출 횟수	직렬화로 감소된 함수 호출 횟수	감소된 비율(%)
7365	2392	32.5

Table 13: 코드 직렬화 미적용코드 수행 시간

쓰레드 개수	직렬화 미적용(ms)
25	12195628
50	13544528
75	13337064
100	13887019

Table 14: 코드 직렬화 적용코드 수행 시간

쓰레드 개수	직렬화 적용(ms)
25	12755048
50	13728037
75	13893552
100	14456961

Table 15: 코드 직렬화 적용코드 수행 시간 비교

쓰레드 개수	속도차(ms)	속도향상(%)
25	-559420	-4.4
50	-183509.4	-1.3
75	-556487.6	-4
100	-569942	-3.9

## 5 결론

최근 인공지능(Artificial Intelligence, AI) 중 딥러닝기술의 발달함에 따라 GPGPU와 병렬처리기술이 발달하고 있다. 인공지능과 같이 병렬처리 구간이 많이 나타나는 분야는 GPGPU의 활용도가 더욱 높아지나, 암호와 같이 병렬화 지수가 낮은 경우 GPGPU의 성능이 높게 나오지 못하고 있다. 특히 암호기술은 이러한 병렬처리구간을 줄이기 위해 노력하고 있으며, salsa와 같은 암호화는 병렬처리구간이 거의 없어 대규모 처리가 아닌 경우 병렬처리를 할 수 없다. 대규모 서버와 같은 환경에서 보안프로토콜을 사용하는 경우 단순 암호 연산이 아닌 GPGPU를 활용하는 병렬처리 기술을 이용하여 암호연산 효율을 올릴 수 있으며, 이러한 경우 많은 데이터를 빠르게 처리해야 하므로 지금까지 연구한 기법을 적용한다면 좀 더 빠르고 쾌적한 컴퓨팅 환경을 제공할 수 있다. 본 논문에선 Curve25519에 대하여 병렬처리 가능 구간에 대한 분석 및 해당 구간에 대한 성능 향상 방안을 제시하였다. 필드 곱셈에서 유휴 쓰레드를 줄이는 방법으로 512 만 번의 곱셈 연산 시 약 17%의 성능 향상이 있었으며, 몽고메리 사다리의 double and add 연산에 대하여 연산 병렬 가능한 구간을 제시하여 약 49%의 성능 향상을 기대할 수 있었다. 또한 멀티 쓰레딩, 루프 언롤링, 코드 직렬화와 같은 GPGPU 성능 튜닝을 통한 성능 향상을 기대할 수 있었다.

## References

- [1] Daniel J. Bernstein, "Curve25519: new Diffie-Hellman speed records," Feb, 2006.
- [2] Daniel J. Bernstein, Tanja Lange, "Security Dangers of the NIST curves", May, 2013.
- [3] Mark Harris, "Optimizing Parallel Reduction in CUDA", NVIDIA Developer Technology
- [4] Kishore Kothapalli Rishabh Mukherjee Suhail Rehman, Suryakant Patidar P. J. Narayanan Kannan Srinathan, "A Performance Prediction Model for the CUDA GPGPU Platform", International Institute of Information Technology, Hyderabad. Dec, 2009
- [5] Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, Senior Member, IEEE, and Jiwu Jing, "An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration", IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, VOL. 12, NO. 1, JANUARY 2017 111-122
- [6] Robert Szerwinski and Tim Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography", Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany, 2008
- [7] 김정우, "Parallel implementations of cryptographic algorithms on GPU", 서울대학교, Feb, 2013

- [8] 윤은준, (2012). 군사 목적의 대용량 자료 암호화를 위한 ECC 기반 암호시스템. 제어로봇시스템학회 학술대회 논문집, 701-705.
- [9] Seo H. et al. (2014) Parallel Implementations of LEA. In: Lee HS., Han DG. (eds) Information Security and Cryptology – ICISC 2013. ICISC 2013.
- [10] Seo H. et al. (2017) Parallel Implementations of LEA, Revisited. In: Choi D., Guilley S. (eds) Information Security Applications. WISA 2016.

## A 약어

*GPU : Graphic Processing Unit*

*GPGPU : General Purpose computing Graphic Processing Unit*

*CUDA : Compute Unified Device Architecture*

*OpenCL : Open Computing Language*

*NIST : National Institute of Standard and Technology*

*NSA : National Security Agency*

*ECC : Elliptic Curve Cryptography*

*MHz : megahertz*

*DRBG : deterministic Random Bit Generator*

*ECDLP : Elliptic Curve Discrete Logarithm Problem*

*ECDH : Elliptic Curve Diffie Hellman*

*DSA : Digital Signature Algorithm*

*GF : Galois field*

*AI : Artificial Intelligence*

*ALU : Arithmetic and Logic Unit*

## B 초록