Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.3, Jun. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.3.481

다중 사용자 환경에서 효과적인 키 교환을 위한 GPU 기반의 NTRU 고속구현

성 효 은,[1†] 김 예 원,[1] 염 용 진,[2‡] 강 주 성[2]
[1]국민대학교 금융정보보안학과 (대학원생), [2]국민대학교 정보보안암호수학과 (교수)

Accelerated Implementation of NTRU on GPU for Efficient Key Exchange in Multi-Client Environment

Hyoeun Seong,[1†] Yewon Kim,[1] Yongjin Yeom,[2‡] Ju-Sung Kang[2]
[1]Dept. of Financial Information Security, Kookmin University
(Graduate student),
[2]Dept. of Information Security, Cryptology and Mathematics, Kookmin University
(Professor)

요 약

대규모 양자컴퓨팅 기술의 실현을 앞둔 현재 공개키 암호 시스템을 양자내성을 가진 암호 시스템으로 전환하는 것은 필수적이다. 미국 국립표준기술연구소 NIST는 양자내성암호(Post-Quantum Cryptography, PQC)를 표준화하기 위한 공모사업을 추진하고 있으며 인터넷 통신 보안에 주로 사용되는 TLS(Transport Layer Security) 프로토콜에 이러한 양자내성암호를 적용하기 위한 차원의 연구도 활발히 진행되고 있다. 본 논문에서는 병렬화된 양자내성암호 NTRU를 활용하여 TLS 상에서 서버와 다수의 사용자가 세션키를 공유하기 위한 키 교환(key exchange) 시나리오를 제시한다. 또한, GPU를 이용하여 NTRU를 병렬화 및 연산을 고속화하는 방법을 제시하고 서버가 대규모 데이터를 처리해야 하는 환경에서 그 효율성을 분석한다.

ABSTRACT

It is imperative to migrate the current public key cryptosystem to a quantum-resistance system ahead of the realization of large-scale quantum computing technology. The National Institute of Standards and Technology, NIST, is promoting a public standardization project for Post-Quantum Cryptography(PQC) and also many research efforts have been conducted to apply PQC to TLS(Transport Layer Security) protocols, which are used for Internet communication security. In this paper, we propose a scenario in which a server and multi-clients share session keys on TLS by using the parallelized NTRU which is PQC in the key exchange process. In addition, we propose a method of accelerating NTRU using GPU and analyze its efficiency in an environment where a server needs to process large-scale data simultaneously.

Keywords: Post-Quantum Cryptography, Key Exchange Protocol, NTRU, GPU, CUDA

Received(04. 23. 2021), Modified(05. 24. 2021), Accepted(05. 24. 2021)

† 주저자, she000@kookmin.ac.kr
‡ 교신저자, salt@kookmin.ac.kr(Corresponding author)

I. 서 론

양자컴퓨팅 기술의 발달로 인한 대규모 양자컴퓨터의 실현은 현재 보안 시스템 전반에 사용되고 있는 공개키 암호 시스템에 위협이 될 것으로 예상된다. 소인수 분해와 이산로그 문제에 기반을 둔 현재의 공개키 암호는 1994년 Shor의 양자 알고리즘[1]이 제안되며 다항 시간 내에 해결 가능하다는 것이 증명되었다. 과거에는 대규모 양자컴퓨터의 물리적 구현 가능성이 명확하지 않았으나 현재 전문가들은 향후 20년 이내에 양자컴퓨터가 구축될 것으로 예상한다. 워털루 대학의 Mosca 교수는 양자컴퓨터를 이용한 공격으로 2031년 현재의 공개키 암호 시스템이 깨질 확률을 1/2 이상으로 예측한다[2]. 따라서 양자컴퓨터 기술을 이용한 공격에도 안전성을 보장받을 수 있는 양자내성암호로의 전환은 필수적이며, 현재 공개키 암호 인프라를 배포하는데 거의 20년이 소요된 것을 고려한다면 시급한 문제로 보인다.

미국 국립표준기술연구소 NIST는 현재 이러한 양자내성암호에 대한 표준화를 위하여 서명(signature) 알고리즘과 공개키 암호(Public Key Encryption, PKE) 및 키 캡슐화 메커니즘(Key Encapsulation Mechanism, KEM)에 대한 공모사업을 진행 중이다[3]. 2016년 공모사업 발표 이후 다양한 문제를 기반으로 하는 64개의 알고리즘이 1라운드 후보 알고리즘으로 공개되었으며 안전성, 성능에 대한 평가를 거쳐 2020년 7개 알고리즘이 3라운드 최종 후보(finalist)로 선정되었다.

Table 1.과 같이 PKE/KEM에 속하는 4개의 최종 후보 알고리즘 중 NTRU, CRYSTALS-KYBER, SABER 3개의 알고리즘이 격자기반 암호(lattice-based cryptography)로 격자기반 기술이 상당한 비율을 차지한다.

Table 2.는 3라운드 최종 후보로 선정된 격자 기반 암호에 대하여 동일한 안전성 수준(security category)을 가지는 파라미터 설정에서 키와 암호문의 크기를 비교한다. NTRU는 동일한 안전성을 제공하는 다른 최종 후보들과 비교하여 KEM 과정에서 전달해야 하는 키와 암호문의 크기가 작다는 점에서 효율성을 가진다. 또한, NTRU는 다른 최종 후보와 비교하여 최고 수준의 성능을 내지는 못하지만 가장 긴 역사를 가지고 오랫동안 분석을 통해 안전성이 검증되어 온 것이 장점이다[4].

Table 1. NIST PQC round 3 finalists

	Signature	PKE/KEM
Lattice	CRYSTALS -DILIGHIUM, FALCON	CRYSTALS -KYBER, NTRU, SABER
Code	-	Classic McEliece
Multivariate	Rainbow	-

Table 2. The size of the key and ciphertext for NIST PQC round 3 finalists (unit: byte)

	NTRU hps4096821	KYBER 1024	Fire SABER
sk	1,590	3,168	3,040
pk	1,230	1,568	1,312
ct	1,230	1,568	1,472

한편 국내에서도 양자내성암호를 IoT와 같은 경량 환경에 적용하는 것과 인터넷 통신 보안 프로토콜인 TLS에 적용하기 위한 차원의 연구를 진행 중이다. [5]는 경량 환경에 적합한 TLS 라이브러리 mbedTLS 상에서 파라미터 데이터가 큰 격자기반 암호를 적용하기 위해 Handshake Message Fragmentation 기능을 지원하는 구현 방식을 제안하였다. [6]에서는 NIST의 공모사업 2라운드 후보 중 코드기반 암호(code-based cryptography) 7가지를 비교하고 암호화 횟수를 최소화하는 등 경량 환경에 적합한 프로토콜을 설계하여 제안하였다. 또한, [7]에는 격자기반 암호를 이용한 인증키 교환(AKE) 프로토콜 7가지에 대하여 기반문제, 안전성 분석 기법, 구현 효율성 등을 비교한 결과가 제시되었다.

본 논문에서는 3라운드 최종 후보로 선정된 주요 격자기반 암호 중 NTRU를 활용하여 TLS1.3 프로토콜 상에서 서버와 다수의 사용자 간의 효율적인 키 교환 방법을 제시한다. 다수의 사용자가 서버에 접속하고자 할 때, 서버는 많은 양의 키 교환 과정을 한번에 처리해야 한다. 이러한 환경을 고려하여 서버가 NTRU KEM을 병렬로 실행하여 다수의 사용자와 키 교환하는 시나리오를 제시하고, 병렬화 및 고속화된 NTRU의 성능향상 결과를 분석한다. 2장에서는 NTRU의 구조와 이를 고속화하기 위한 GPU 병렬 구현 플랫폼인 CUDA에 대해 설명하고 3장에서는 병렬화 시나리오와 실제 구현 방법을 설명한다. 4장에서는 NTRU의 주요 연산을 고속화하는 방법 및 그 효율성을 분석한다. 마지막으로 3장과 4장의 구현 기법을 통한 NTRU의 전체적인 성능향상 결과를

5장에서 분석하고 이를 최적화하는 방법을 제시한다.

II. 배경 지식

2.1 NTRU 알고리즘

NTRU는 NIST가 진행 중인 양자내성암호 공모사업의 3라운드 최종 후보에 선정된 격자기반 암호 중 하나이다[3]. NTRU는 함께 최종 후보로 선정된 격자기반 암호 KYBER, SABER가 LWE(Learning With Error) 문제를 기반으로 하는 것과 다르게 NTRU 문제를 가정하여 안전성을 보장한다.

NTRU 문제는 격자기반 문제 중 SVP(Shortest Vector Problem) 문제에 속한다[8]. SVP 문제는 벡터 B를 기저로 생성되는 격자 공간(lattice space) $L(B)$에서 식(1)과 같이 0이 아닌 가장 짧은 벡터 v를 찾는 문제이다.

$$\|v\| = \min\{\|v\| : v \in L(B)\}. \quad (1)$$

따라서 NTRU 문제는 공개키 h가 주어질 때 h를 이용하여 정의된 격자 공간 L_h에서 가장 짧은 벡터 (f,g)를 찾는 것이 어렵다는 것을 가정한다. 이때, f와 g는 NTRU의 키생성 과정에서 생성되는 다항식으로 개인키를 구성하는 요소가 된다.

NTRU의 DPKE(Deterministic Public Key Encryption)는 OW-CPA(One-Wayness against Chosen Plaintext Attack)을 제공하며 이에 기반하여 설계된 KEM은 IND-CCA2(Indistinguishability under Chosen Ciphertext Attack 2)를 제공한다. 또한, NIST가 양자내성암호의 안전성을 평가하기 위하여 설정한 안전성 수준에 대하여 보안 강도 1, 3, 5에 해당하는 파라미터 세트를 가진다[9].

Table 3.은 NTRU의 NTRU-hps에 속하는 3개의 파라미터 세트와 NTRU-hrss에 속하는 1개의 파라미터 세트에 대해 설명한다. NTRU 알고리즘의 연산은 다항식 환(polynomial ring) 상에서 이루어지는 것을 특징으로 한다. Table 3.의 n은 연산에 사용되는 다항식의 최대 차수를 결정하는 파라미터로 연산에 사용되는 다항식 환은 다음과 같다.

Table 3. Parameter sets and security categories of NTRU-KEM

	NTRU-hps			NTRU-hrss
n	509	677	821	701
p	3	3	3	3
q	2,048	2,048	4,096	8,192
security	1	3	5	3

$$R = Z[x]/(x^n - 1),\ S = Z[x]/(\frac{x^n - 1}{x - 1}). \quad (2)$$

예를 들어, $n = 821$인 파라미터 세트의 경우, 최대 820차 다항식을 연산에 사용하게 된다. 이때, 파라미터 p와 q는 다항식의 계수를 모듈러(modular)하는 수로 R의 다항식들의 계수를 q로 모듈러 하여 생성된 다항식 환을 R_q와 같이 표기한다. NTRU의 다항식 샘플링은 다항식 환 $S_p = Z_p[x]/(\frac{x^n - 1}{x - 1})$에서 실행된다.

NTRU의 KEM은 이러한 다항식 연산을 사용하여 키를 공유하는 메커니즘으로 키 생성 과정, 캡슐화(encapsulation) 과정, 그리고 역캡슐화(decapsulation) 과정으로 나뉜다. Fig. 1.은 NTRU의 KEM 과정에서 사용되는 다항식과 키의 전달과정을 도식화한 것이다. 캡슐화 과정에선 두 다항식 r과 m을 샘플링한 후 해시함수를 사용하여 공유키 k를 생성하고 공개키 h로 두 다항식을 암호화

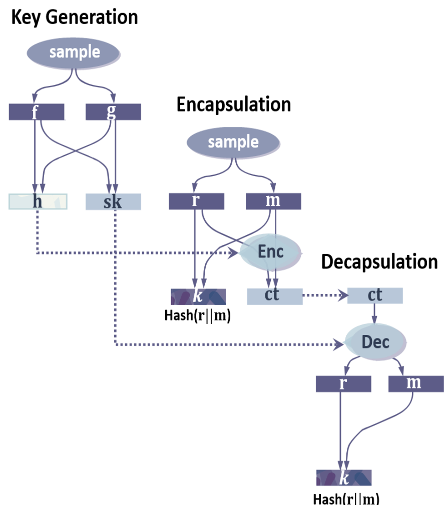

Fig. 1. Description of NTRU KEM

하여 전달한다. 역캡슐화 과정에선 전달받은 암호문을 개인키 sk로 복호화하여 r과 m을 얻고 해시함수를 사용하여 캡슐화와 동일하게 공유키 k를 생성한다. 이를 통해 캡슐화와 역캡슐화를 실행하는 개체는 동일한 키 k를 공유할 수 있다. 각 과정의 주요 연산은 다음과 같다.

- **키 생성**($seed$)
  - 다항식 샘플링: $(f,g) \leftarrow \mathrm{Sample}_{fg}(seed)$
  - 공개키 생성: $h = pf_q * g \ (\mathrm{mod}\, q)$
  - 개인키 생성: $sk = f_p \| f \| h_q \| s \ (\mathrm{mod}\, q)$
  - 반환: (sk, h)
- **캡슐화**(h)
  - 다항식 샘플링: $(r,m) \leftarrow \mathrm{Sample}_{rm}(\mathrm{seed})$
  - 공유키 생성: $k = \mathrm{Hash}(r,m)$
  - 암호화: $ct = r * h + m \ (\mathrm{mod}\, q)$
  - 반환: (ct, k)
- **역캡슐화**(sk, ct)
  - 메시지 복호화: $m = f_p * (f * ct) \ (\mathrm{mod}\, p)$
  - 다항식 복호화: $r = (ct - m) * h_q \ (\mathrm{mod}\, q)$
  - 공유키 생성: $k_1 = \mathrm{Hash}(r,m)$
  - prf키 생성: $k_2 = \mathrm{Hash}(ct,s)$
  - 반환: $k = \mathrm{Cmov}(k_1, k_2)$

f,g,r,m은 다항식 환 S상의 다항식으로 샘플링 함수 Sample_{fg}를 통해 랜덤하게 생성된다. f_q, f_p, h_q는 f, h의 다항식 환 S상의 역원(inverse)으로 역원 계산 시 다항식의 계수에 모듈러 q를 사용하면 f_q와 같이 표기한다. 키 생성 과정의 다항식의 역원을 구하는 연산은 일반적으로 사용되는 확장 유클리드 알고리즘 대신 좀 더 효율적으로 계산이 가능한 'almost inverse 알고리즘'을 사용한다[10]. 또한, 공유키를 구하는 해시(hash)함수로는 256비트를 출력하는 sha3-256 연산을 사용한다. 기호*는 다항식 환상의 다항식 곱셈 연산을 의미하며 현재 NIST 공모사업에 제안된 NTRU에서는 합성곱(convolution multiplication) 알고리즘을 사용하여 다항식 연산을 실행한다. Cmov함수는 r,m이 제대로 복호화된 경우에만 r,m으로 생성된 키 k_1를 반환하는 역할을 한다.

2.2 CUDA 프로그래밍

GPU는 다량의 데이터를 병렬로 처리하는데 적합한 구조를 가진다. NVIDIA 사(社)에서 2006년 소개한 CUDA는 GPU를 이용한 병렬 컴퓨팅(parallel computing)을 C와 같은 언어상에서 구현할 수 있도록 하여 병렬화가 필요한 문제에 관해 CPU보다 효율적으로 해결할 수 있도록 하는 플랫폼이다[11]. CUDA는 사용자가 커널함수(kernel function)를 정의함으로써 병렬 컴퓨팅을 가능하게 한다. CUDA 프로그래밍에서 호스트(host)는 보통 CPU를 의미하고 디바이스(device)는 GPU를 의미한다. 호스트에서 정의된 커널함수를 호출하면 디바이스에서 커널함수가 병렬로 실행된다.

GPU가 가지는 수천 개의 CUDA 코어(core)들은 각 독립적인 멀티프로세서(multiprocessors, MP)에 할당되어 있다. CUDA 프로그래밍에서 사용자가 사용할 수 있는 메모리는 크게 디바이스 내에서 사용 가능한 온칩(on-chip) 메모리와 호스트와 디바이스가 모두 접근 가능한 오프칩(off-chip) 메모리로 나뉜다. 글로벌 메모리(global memory)는 오프칩 메모리로 모든 CUDA 블록(block) 내의 스레드(thread)가 접근할 수 있다. 한 블록 내의 스레드들이 공유하는 공유 메모리(shared memory)와 스레드의 고유 메모리인 레지스터(register)는 온칩 메모리에 해당하며 커널함수의 선언에 따라 사용량이 결정된다.

커널함수를 선언 후 런칭(launching)할 때, 커널함수의 파라미터로 생성할 블록과 블록 내의 스레드의 수를 결정한다. 이때, 공유 메모리와 레지스터의 사용량, 그리고 디바이스에서 한 번에 동작하는 스레드 단위를 의미하는 워프(warp)의 제약을 고려하여 최대한 많은 워프가 동시에 작업을 실행할 수 있도록 하는 것이 중요하다. GPU마다 구조와 가지고 있는 메모리 자원이 다르므로 커널함수를 호출할 때 블록과 스레드 수를 적절히 조절해야만 높은 성능으로 프로그램을 병렬화하는 것이 가능하다.

2.3 용어 설명

Table 4.는 본 논문의 3, 4, 5장에서 사용할 기호와 그 의미를 정리한 것이다.

Table 4. Notations

N	number of executions in kernel function (in Section 3.2, encapsulation or decapsulation executions, in Section 4.3, operation executions)
N_2	block tasks for one execution (ex. for a polynomial multiplication, it means number of coefficients of a polynomial)
B	number of blocks generated in kernel function
T	total number of threads in a block

III. NTRU KEM 병렬화

본 장에서는 2장에서 설명한 CUDA 블록을 사용하여 NTRU KEM을 병렬화하는 방법에 관해 설명한다. 3.1절에서는 NIST의 표준화 이후 양자내성암호가 적용될 TLS1.3 프로토콜에 관해 설명하고 3.2절에서는 이를 위해 제안하는 병렬화 시나리오를 설명한다.

3.1 TLS1.3 프로토콜

TLS는 두 개체 간의 암호화된 데이터 전송을 가능하게 하는 프로토콜로 국제 인터넷 표준화 기구(IETF)에서 인터넷 통신 보안을 위한 표준으로 선정하였다. 현재 TLS1.3은 핸드셰이크를 통한 키 교환(key exchange) 과정에 ECDH 알고리즘을 사용하고 있다. NIST의 표준화 이후 양자내성암호가 이를 대체할 것으로 보이며 이에 따라 표준화 후보들의 TLS1.3 프로토콜에 대한 적합성 또한 다방면으로 연구되고 있다.

NTRU는 TLS1.3 프로토콜의 사양을 변경하지 않고 키 교환 과정에 적용 가능하였으며, 기존의 알고리즘인 ECDH와 함께 사용하는 하이브리드(hybrid) 모드와 NTRU 만을 단독으로 사용하는 모드에 모두 적합하다는 결과를 보였다[12]. 또한, ICMC 2020에서는 NTRU가 레이턴시(latency), 키사이즈 등을 고려할 때 다른 2라운드 후보였던 SIKE와 비교하여 TLS1.3 프로토콜에 적합하다는 결과가 발표되었다[13].

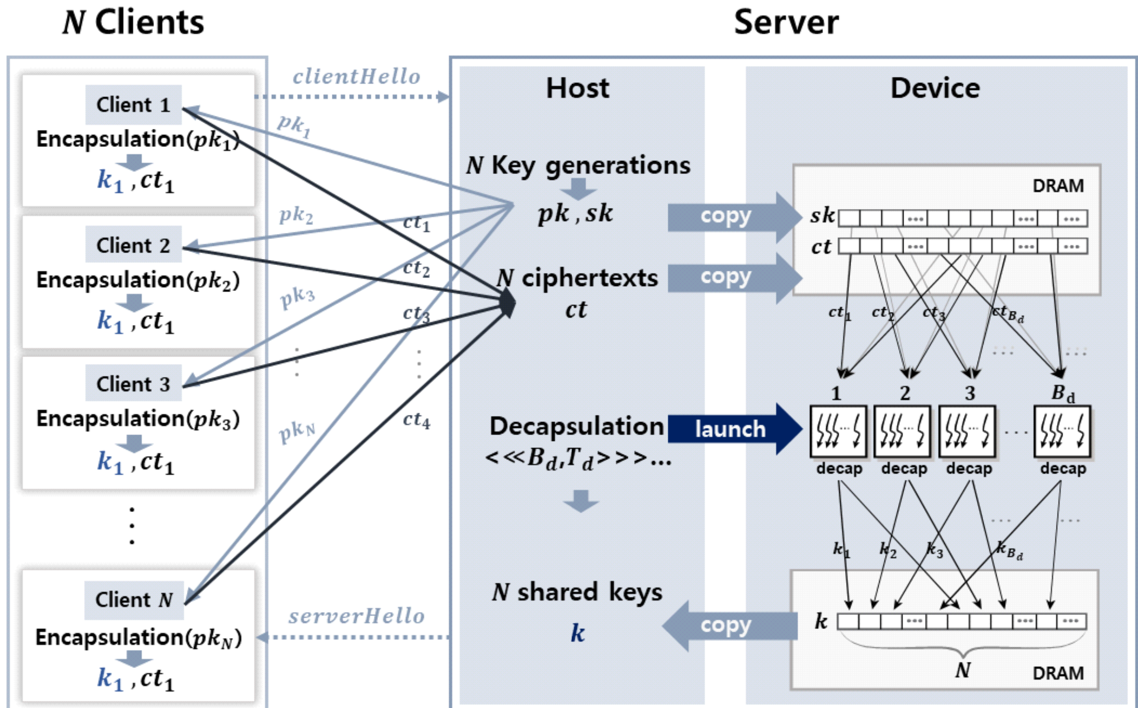

Fig. 2. Parallel execution of decapsulations for key exchange with N clients

3.2 병렬화 시나리오

TLS1.3 프로토콜을 통해 사용자(client)는 서버(server)와의 암호화된 통신에 사용할 키를 공유할 수 있다. N명의 사용자가 동시에 서버와 키를 공유하고자 할 때 서버에서 이를 차례대로 처리하게 되면 세션을 맺는 과정이 지연될 수 있다. 따라서 본 논문에서는 NTRU KEM이 TLS1.3 프로토콜의 키 교환 과정에 적용되는 방법 중 Fig.2.와 같은 병렬화 시나리오를 제안한다.

- **키 생성 과정**
  서버에 접속하고자 하는 i번째 사용자는 서버에 `ClientHello` 메시지를 통해 알고리즘과 파라미터 정보를 전달한다($1 \le i \le N$). 서버는 키 생성 과정을 통해 N개의 공개키와 개인키 쌍 (pk_i, sk_i)을 생성하고 각 사용자에게 pk_i를 전달한다.
- **캡슐화 과정**
  각 사용자가 캡슐화 과정을 실행한다. 전달받은 pk_i로 암호화된 암호문 ct_i와 공유키 k_i를 생성하고 ct_i를 서버에게 전달한다.
- **역캡슐화 과정**
  서버는 전달받은 N개의 ct_i 각각에 대하여 생성한 sk_i로 역캡슐화 과정을 실행한다. 이때, 역캡슐화 과정은 N번 차례대로 실행되는 것이 아닌 커널함수로 선언되어 디바이스에서 병렬로 실행된다. 병렬화된 역캡슐화 과정의 결과로 서버는 N명의 사용자와 k_i를 공유하게 된다.

키 교환 과정이 완료되면 서버는 각 사용자에게 `ServerHello` 메시지를 보내고 이후로 각 사용자는 공유키 k_i를 이용하여 서버와 암호화된 데이터를 주고받게 된다.

반대로 서버가 캡슐화를 병렬로 실행하고 각 사용자가 역캡슐화를 실행하는 시나리오는 다음과 같다.

- **키 생성 과정**
  i번째 사용자는 키 생성 과정을 통해 공개키와 개인키 쌍(pk_i, sk_i)을 생성한 뒤 `ClientHello` 메시지를 통해 pk_i를 서버에게 전달한다 $(1 \le i \le N)$.
- **캡슐화 과정**
  서버는 각 사용자에게 전달받은 N개의 pk_i로 캡슐화 과정을 실행하여 암호문 ct_i와 공유키 k_i를 생성한다. 이때, 캡슐화 과정은 커널함수로 선언되어 디바이스에서 병렬로 실행된다. 이후 생성된 N개의 암호문 ct_i를 `ServerHello` 메시지를 통해 각 사용자에게 전달한다.
- **역캡슐화 과정**
  사용자가 각각 sk_i로 역캡슐화 과정을 실행하여 전달받은 ct_i를 복호화하고 k_i를 공유하게 된다.

본 논문에선 이러한 시나리오를 실행하기 위해 커널함수 런칭 시 각 블록이 캡슐화 과정 또는 역캡슐화 과정을 실행하도록 하였다. 즉, 다수의 블록이 동일한 과정을 각각 독립적으로 실행하여 한 번에 런칭된 블록의 개수만큼 공유키를 생성하게 된다. 예를 들어 첫 번째 시나리오에서 서버는 N개의 공유키를 생성하기 위해 디바이스의 글로벌 메모리에 N개의 공유키와 암호문의 메모리를 할당한 후 역캡슐화 커널을 호출한다.

역캡슐화 커널 런칭 시 생성되는 블록의 수를 B_d로 설정하였다고 하자. 역캡슐화를 총 N번 실행하기 위해 각 블록은 그리드 내에서 가지는 블록 고유의 인덱스(index)에 따라 Fig.3.과 같이 반복문을 실행한다. 이때, 하나의 블록이 반복문을 실행하는 최대 횟수는 $W_1 = \lceil N/B_d \rceil$ 다. 즉, 하나의 블록은 역캡슐화를 반복하여 W_1개의 공유키를 생성하고 글로벌 메모리의 지정된 주소에 접근하여 저장한다.

캡슐화 과정 역시 생성된 블록의 수 B_e에 따라

Algorithm 1 Decapsulation kernel function with B_d blocks (T_d threads per a block)

Input: N Private keys sk, Ciphertexts ct
Output: N Shared keys k

```
1: b_id = blockIdx.x                // block index
2: while(b_id < N) do
3:     (k[b_id]) ← Decapsulation(sk[b_id], ct[b_id])
4:       //decapsulation with T_d threads in
5:        a CUDA block
6:     b_id += B_d
7: end while
```

Fig. 3. Parallel execution of N decapsulations with multiple blocks on device

각 블록이 반복문을 병렬로 실행한다. 역캡슐화 과정과 마찬가지로 각 블록은 $W_1 = \lceil N/B_e \rceil$ 번 캡슐화를 반복하게 된다.

IV. NTRU 연산 고속화

본 장에서는 NTRU 캡슐화 또는 역캡슐화 과정의 구성요소 중 병렬화가 가능한 연산을 분석하고 이를 병렬화하여 고속화한다. 4.1절에서는 NTRU의 구성요소를 분석하여 병렬화할 연산을 결정한다. 4.2절은 블록 내에서 다수의 스레드를 이용한 연산의 병렬화 기법을 설명하고 4.3절에서 그 효율성을 분석한다.

4.1 NTRU 구성요소 분석

NTRU 캡슐화 과정과 역캡슐화 과정의 주요 연산을 파악하기 위해 연산 별 소요시간을 측정한 결과는 Table 5.와 같다. 실험에는 저자가 NIST 공모사업의 3라운드에 제공한 소스코드 중 최적화 버전을 사용하였으며 Intel Core i7-4790K CPU 상에서 소요시간을 측정하였다.

Table 5.는 캡슐화 과정과 역캡슐화 과정을 각각 $N=10{,}000$번씩 반복하였을 때 소요시간이 $1ms$ (millisecond) 이상 측정되는 연산들을 기재한 것으로 비교적 짧은 시간이 소요되는 인코딩(encoding) 연산들은 제외하였다. Rq-mul과 같이 여러 번 호출되는 연산은 ×(호출횟수)를 표기하였다.

Table 5. Execution time for NTRU-hps4096821 on CPU ($N=10{,}000$)

Encapsulation		4,704 ms	(100%)
randombytes		651 ms	(13.8%)
sample-iid		8 ms	(0.1%)
sample-fixed		382 ms	(8.1%)
Rq-mul		**3,534 ms**	**(75.1%)**
sha3-256		52 ms	(1.1%)
Decapsulation		11,153 ms	(100%)
Rq-mul	×3	**10,714 ms**	**(95.5%)**
mod$(3, x^n - 1)$	×4	103 ms	(0.9%)
sha3-256	×2	192 ms	(2.0%)
S3 frombytes		3 ms	(0.0%)
cmov		1 ms	(0.0%)

측정 결과 다항식 곱셈 연산인 Rq-mul이 캡슐화 과정의 전체 소요시간 중 약 75%, 역캡슐화 과정의 전체 소요시간 중 약 95%를 차지하였다. 이에 따라 다항식 곱셈 연산의 성능이 150배 향상될 경우 캡슐화 과정 전체성능이 약 3.9배, 역캡슐화 과정 전체 성능이 약 17.7배 향상될 것으로 추정하였다. 또한, 캡슐화 과정과 역캡슐화 과정의 고속화를 위해 가장 많은 시간이 소요되는 다항식 곱셈 연산에 대한 고속화가 필요할 것으로 분석하였다.

다항식 곱셈 연산 외에도 소요시간과 호출되는 빈도를 고려하여 병렬 고속화 가능성을 분석할 대상으로 선정한 주요 연산은 다음과 같다.

- **캡슐화 과정**
  : Rq-mul, randombytes, sample-iid, sample-fixed, sha3-256
- **역캡슐화 과정**
  : Rq-mul, mod$(3, x^n - 1)$, sha3-256

4.2 연산 고속화 방법

4.1절에서 분석한 결과와 같이 NTRU의 캡슐화 및 역캡슐화 과정은 다항식 곱셈 연산에 대부분 시간이 소요된다. NTRU의 다항식 환은 NTT 변환을 사용한 다항식 곱셈 연산을 할 수 없다. NTRU의 다항식 환인 $R = Z[x]/(x^n - 1)$ 상에서 두 다항식 $a(x) = \sum_{i=0}^{n-1} a_i x^i$와 $b(x) = \sum_{i=0}^{n-1} b_i x^i$의 다항식 곱셈 연산은 일반적으로 합성곱 알고리즘을 사용하여 다음과 같은 식으로 계산된다.

$$c(x) = \sum_{i=0}^{n-1} c_i x^i = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j x^{(i+j \bmod n)}. \quad (3)$$

이때, 연산 결과인 다항식 $c(x)$의 i번째 계수 c_i는 $\sum_{j=0}^{n-1} a_j b_{(i-j \bmod n)}$로 계산된다. 즉, $c(x)$의 각 계수들은 차례대로 계산될 필요 없이 각각 독립적으로 계산 가능하다. 따라서 [14]에서는 이러한 합성곱 알고리즘에 대하여 여러 개의 스레드가 협력하여 계산하도록 하는 병렬 고속화 방법을 제안하였다. [14]에서 고속화한 서명 알고리즘 NTRU-MLS는 NTRU와 유사한 형태의 다항식 환을 사용하였다.

따라서 본 논문에서는 NTRU에 해당 방법을 적용할 수 있을 것으로 예상하여 다항식 곱셈 연산을 비롯한 주요 연산에 대해 고속화 가능성을 파악하였다. 본 논문의 실험에선 블록당 스레드의 개수 T가 다항식 항의 개수 n보다 작을 경우를 고려하여 Fig.4.와 같이 while 반복문을 사용한 연산 실행 방법을 사용하였다.

Fig.4.에서 같이 블록 내의 스레드는 n개의 다항식 계수를 계산하기 위해 동일한 작업을 병렬로 실행한다. 블록 내의 스레드의 수를 T, 스레드가 협력하여 실행해야 할 전체 작업을 $N_2 = n$으로 표기할 때 하나의 스레드가 반복문을 실행해야 하는 최대 횟수는 $W_2 = \lceil N_2/T \rceil$ 이다. 다시 말해, 어떤 스레드 고유의 인덱스가 t_{id}라면 다항식 $c(x)$의 $t_{id}, t_{id}+T$, ..., $t_{id}+(W_2-1)T$번째의 계수를 계산하여 다항식 곱셈 연산을 실행한다.

이러한 방식은 4.1절에서 분석한 NTRU의 주요 연산 중 결과로 출력되는 다항식 각 계수 또는 바이트 열의 각 원소가 독립적으로 계산 가능한 연산의 경우 적용이 가능하다. 예를 들어 난수 바이트 열을 생성하는 randombytes 연산의 경우 각 바이트를 차례대로 생성할 필요 없이 병렬로 생성할 수 있다.

Fig.5.와 같이 본 논문에서는 randombytes 연산에 CUDA에서 제공하는 난수발생기인 curand 함수를 사용하여 각 바이트를 독립적으로 생성하였다. 각 바이트마다 다른 씨드(seed)로 생성하기 위해 스레드 고유의 인덱스와 블록의 고유 인덱스를 사용하여 함수를 실행하였다. 이때, 블록 내의 스레드들이 협력하여 실행해야 할 전체 작업은 $N_2 = \text{len}_{seed}$가 되고 이를 위해 하나의 스레드가 반복해야 할 실행의 수는 $W_2 = \lceil \text{len}_{seed}/\text{T} \rceil$가 된다. S_p상에서 다항식을 샘플링하는 sample-iid 연산도 마찬가지이다. sample-iid는 각 바이트에 모듈러 3을 취하여 다항식의 계수를 샘플링 한다. 각 스레드는 인덱스에 따라 계수를 계산하며 이때, 스레드가 협력하여 실행해야 할 전체 작업 $N_2 = n-1$, 각 스레드가 반복문을 실행해야 할 횟수는 $W_2 = \lceil n-1/T \rceil$ 가 된다.

그러나 계수에 조건을 가지는 다항식 샘플링 연산 sample-fixed는 완전한 병렬화가 불가능하다. sample-fixed로 생성된 다항식은 계수 중 -1과 1의 개수가 NTRU 파라미터 q에 의해 결정되어 있다. 따라서 각 계수를 독립적으로 생성하지 못하고 차례대로 계산해야 하는 함수가 연산 내에 포함된다. sha3-256 연산 또한 입력된 각 바이트에 대하여 순차적인 업데이트가 필요한 함수를 포함하여 완전한 병렬화가 불가능하다. 이처럼 병렬화가 불가능한 함수는 하나의 스레드가 차례대로 실행하도록 하였다.

Algorithm 2 Rq-mul

Input: Polynomials a, b, Degree n
Output: Polynomial c

```
1:  procedure Rq-mul(n, c, a, b)
2:    t_id ← threadIdx.x            // thread index
3:    while (t_id < n) do
4:      for i ← 0 to n do
5:          c[t_id] += a[i] × b[(t_id − i) mod n]
6:      end for
7:      t_id += T
8:      // T: # of total threads in a block
9:    end while
10: end procedure
```

Fig. 4. Parallelized polynomial multiplication with T threads

Algorithm 3 randombytes

Input: Random bytes length len_{seed}
Output: Random bytes $seed$

```
1: procedure randombytes(len_seed, seed)
2:   t_id ← threadIdx.x            // thread index
3:   while (t_id < len_seed) do
4:       seed[t_id] = curand(&state[t_id + blockDim.x
5:                   ×blockIdx.x])
6:     t_id += T
7:     // T: # of total threads in a block
8:   end while
9: end procedure
```

Fig. 5. Parallelized random bytes generator with T threads

4.2.1 캡슐화 커널

하나의 블록이 캡슐화 과정을 실행할 때 각 연산은 블록 내의 스레드가 협력하여 실행할 수도 있고 지정된 번호의 스레드 하나가 차례대로 실행할 수도 있다. 이때, 블록 내의 스레드가 협력하여 실행하는 연산의 경우 연산에 사용되는 다항식 또는 바이트 열(byte array)을 블록 내의 스레드들이 공유할 수 있도록 공유 메모리에 선언하였다.

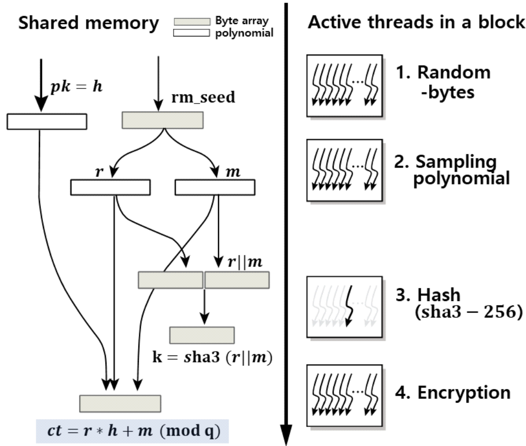

Fig. 6. Shared memory usage and threads in a block that executes NTRU encapsulation

Fig.6.은 캡슐화 커널의 각 과정에서 블록의 공유 메모리의 사용과 블록 내에서 활성화되는 스레드를 도식화한 것이다. 캡슐화 커널의 경우 다항식 곱셈 연산은 한번 호출되며 다항식 샘플링 연산 sample-fixed와 해시함수 sha3-256 연산을 제외한 대부분의 연산이 병렬화 가능하였다. 병렬화할 수 없는 연산의 경우 지정된 하나의 스레드가 실행하도록 하였다.

4.2.2 역캡슐화 커널

Fig.7.은 역캡슐화 커널의 각 과정이 진행됨에 따라 블록 내 공유 메모리의 사용과 블록 내에서 활성화되는 스레드를 도식화한 것이다. 역캡슐화 커널에서 다항식 곱셈 연산은 총 3번 호출된다. 이는 4.1절에서 분석한 것과 같이 역캡슐화 과정이 캡슐화 과정보다 많은 시간이 소요되는 주요인으로 다항식 곱셈 연산의 병렬화 효율성이 높을수록 역캡슐화 커널 전체의 성능향상이 캡슐화 커널보다 클 것으로 예상하였다. 마찬가지로 병렬화할 수 없는 sha3-256 연산은 하나의 지정된 스레드가 실행하도록 하였고 나머지 연산들은 스레드가 협력하여 병렬로 실행하도록 하였다.

또한, 역캡슐화 커널은 메시지 복호화 후 마지막 단계에 해시함수 sha3-256 연산을 두 번 사용한다. sha3-256과 같이 하나의 스레드가 실행하는 연산은 작업을 실행하지 않는 상태인 유휴자원(idle process)의 점유율이 높아진다. 따라서 유휴자원의

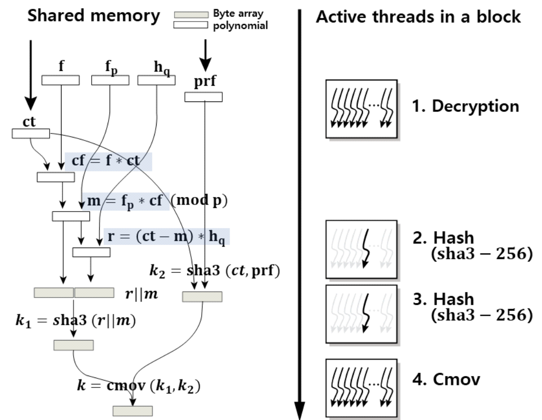

Fig. 7. Shared memory usage and threads in a block that executes NTRU decapsulation

점유율을 감소시키고 병렬화 효율성을 높이기 위해 역캡슐화 커널의 마지막 단계인 sha3-256 연산을 분리하여 다른 커널에서 실행하도록 하는 방법이 있다(이는 5.3절에서 자세히 설명한다). 물론 캡슐화 커널 또한 해당 방법을 적용하는 것이 가능하나 해시함수를 2번 호출하는 역캡슐화 커널에 적용될 때만큼의 성능향상을 보이진 않을 것이다.

4.3 연산 병렬화 효율성 분석

NTRU의 각 연산에 대하여 단일 스레드를 사용하는 것과 T개의 스레드를 사용하여 병렬화한 성능을 비교하기 위해 식(4)와 같은 성능향상률(Speedup)을 계산할 수 있다.

$$\text{Speedup} = \frac{\text{serial execution time}(1\,\text{thread})}{\text{parallelized execution time}(T\ \text{threads})}\ . \quad (4)$$

연산 별로 병렬화 효율성을 확인하기 위해 각 연산을 커널함수로 선언하여 스레드 수를 증가시키며 실험을 진행하였다. 이때, 측정하고자 하는 성능향상률은 실제 연산에 소요된 시간이 스레드에 따라 얼마나 감소하는가이지만 커널함수의 실행시간은 커널함수 호출 시 발생하는 레이턴시와 같이 부가적으로 소요되는 시간을 포함한다. 즉 T개의 스레드를 이용하여 연산을 N번 실행하는 경우 커널함수의 실행시간은 식(5)와 같이 표현할 수 있다.

$$\mathrm{T}_{\mathrm{total}}^{(T,N)} = \mathrm{T}_{\mathrm{runtime}}^{(T,N)} + \mathrm{T}_{\mathrm{overhead}}\,. \tag{5}$$

위의 식(5)에서 $\mathrm{T}_{\mathrm{runtime}}^{(T,N)}$ 은 연산의 실제 실행시간을, $\mathrm{T}_{\mathrm{overhead}}$는 부가적인 소요시간을 의미한다. 이에 의해 연산의 성능향상률은 다음 식(6)으로 표현된다.

$$\mathrm{Speedup} = \frac{\mathrm{T}_{\mathrm{runtime}}^{(1,N)} + \mathrm{T}_{\mathrm{overhead}}}{\mathrm{T}_{\mathrm{runtime}}^{(T,N)} + \mathrm{T}_{\mathrm{overhead}}}\,. \tag{6}$$

따라서 각 연산의 성능향상률은 4.2절에서 설명한 N_2와 소요시간 계산에 영향을 미치는 N, T에 따라 달라질 것이다. 각 연산의 커널함수에서 N_2는 블록이 처리해야 할 작업량으로 연산에 따라 결정되는 값이다. 연산의 실행횟수 N은 너무 작은 값을 사용할 경우 $\mathrm{T}_{\mathrm{runtime}}^{(T,N)}$ 으로 작은 값이 측정되고 $\mathrm{T}_{\mathrm{overhead}}$값의 비율이 커지기 때문에 성능향상률이 높지 않아 병렬화 효과가 미미하다. 따라서 $N=10{,}000$으로 고정된 값을 사용하였다. 마지막으로, 4.2절에서 설명한 커널함수의 공유 메모리와 레지스터 사용량을 고려한 분석 결과 설명한 캡슐화 커널은 768개, 역캡슐화 커널은 640개의 스레드를 블록에 할당 가능하였기 때문에 실험 시 각 연산의 커널함수가 생성할 스레드의 수는 $1 \le T \le 768$으로 설정하였다. 이때, 각 연산은 N_2와 연산의 복잡도에 따라 스레드의 병렬화 효율성이 달라질 것으로 예상하였다. 또한, 연산 내에 병렬화가 어려운 함수를 포함한 경우 연산 도중 제어문 사용에 의한 성능 저하로 병렬화 효율성이 매우 낮을 것으로 예상하였다.

Fig.8-Fig.10.은 각 연산의 T에 따른 성능향상률을 그래프로 나타낸 것으로, 최대 820차 다항식에 대한 연산을 수행하는 NTRU-hps4096821 파라미터 세트상에서 실험한 결과이다. 이때 직선은 병렬화 효율성 100%인 이상적인 경우를 나타낸다.

실험 결과 $N_2=3{,}895$으로 N_2가 연산 중 가장 컸던 `randombytes` 연산과 연산 복잡도가 높은 `Rq-mul` 연산은 비교적 선형에 가까운 그래프를 보였다. 다른 연산의 그래프 역시 유사한 개형을 보였으나 전체를 병렬화할 수 없었던 `sample-fixed` 의 경우 병렬화에 의한 성능향상이 거의 없는 것으로 나타났다. 이처럼 연산 도중 차례대로 실행해야 하는 함수를 포함하는 경우 연산의 일부를 변경하더라도

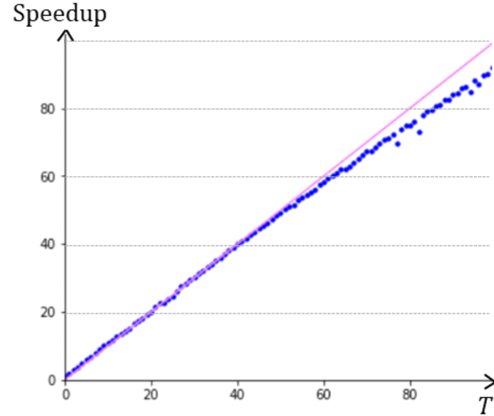

Fig. 8. Speedup of `randombytes` according to the number of threads T

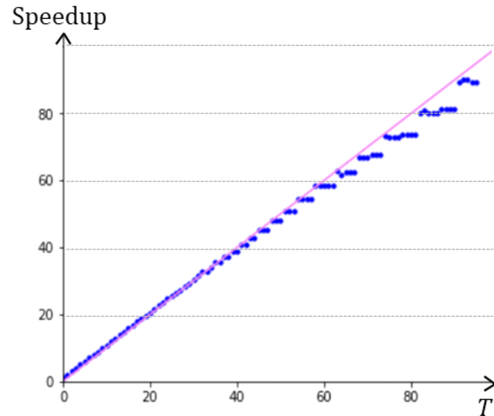

Fig. 9. Speedup of `Rq-mul` according to the number of threads T

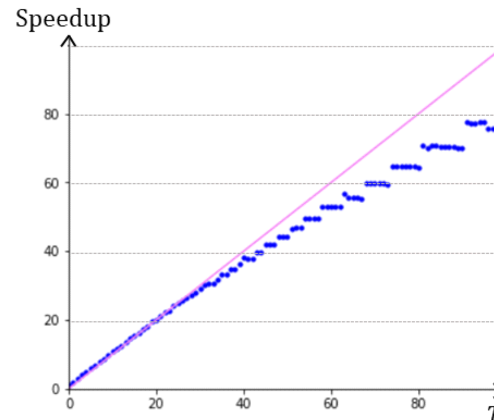

Fig. 10. Speedup of `sample-iid` according to the number of threads T

Table 6. The number of registers per thread and minimum execution time for each operation in NTRU-hps4096821 with 1 CUDA block (N: number of executions, N_2: block tasks for one execution)

	num. of registers	N_2	$N=10,000$	
			time(ms)	Speedup
randombytes	18	3,895	7.8	173.2
sample-iid	16	820	3.1	183.6
sample-fixed	22	-	11662.3	1.0
Rq-mul	58	821	604.3	159.3
mod$(3, x^n-1)$	20	820	3.1	183.6
Lift	19	821	3.5	232.8
+ mod(q, x^n-1)	19	821	2.6	209.4
- mod(q, x^n-1)	19	821	2.7	209.4

병렬화 효율성은 매우 낮을 것으로 보인다.

Table 6.은 NTRU의 캡슐화, 역캡슐화 과정의 주요 연산에 대하여 사용될 것으로 예상하는 레지스터 개수와 병렬화를 통한 최대 성능향상률을 식(4)와 같이 계산하여 정리한 것이다. 이를 통해 전체 캡슐화 커널과 역캡슐화 커널의 연산 병렬화를 통한 성능향상률을 예측할 수 있다. 캡슐화 과정의 전체 소요시간의 약 75%를 차지했던 다항식 곱셈 연산의 경우 최대 159배 성능향상이 가능하였고 13%를 차지하는 연산 randombytes가 최대 173배 성능향상이 가능하였다. 그러나 8%를 차지하는 sample-fixed는 성능향상이 거의 불가능하였다.

따라서 1개의 블록이 캡슐화 커널을 실행할 때 각 연산이 최대로 성능향상될 경우, 약 11배 성능향상이 가능하다. 역캡슐화 과정에서 다항식 곱셈 연산은 전체 소요시간의 95%를 차지하며 약 1%를 차지하는 연산 mod$(3, x^n-1)$는 최대 183배 성능향상이 가능하였다. 나머지 중 2%를 차지하는 해시함수 sha3-256은 병렬화가 불가능하므로 1개의 블록이실행 시 역캡슐화 커널은 최대 약 38배 성능향상이 가능할 것으로 보인다.

그러나 이는 이상적인 수치로 실제 연산들을 조합하여 만들어지는 전체 커널은 예측한 만큼 성능이 향상되지 않을 것이다. 많은 연산을 포함하는 커널은 더 많은 자원을 사용하게 된다. 또한, 커널 내에서 각 연산 후 스레드들을 동일한 준비상태로 만드는 syncthreads()와 같이 부가적인 작업을 실행하기 때문에 연산 고속화만으로 예상한 수치만큼 성능이 향상되지 않을 것이다.

V. 성능분석 및 최적화 방안

본 장에서는 3장과 4장에서 기술한 병렬화 및 고속화 방법을 적용한 캡슐화, 역캡슐화 커널함수의 성능을 분석하고 최적화 방안을 제시한다.

5.1 자원 활용 최적화

GPU를 사용한 병렬구현의 성능은 GPU의 자원을 얼마나 효율적으로 사용하느냐에 따라 달라진다. 멀티프로세서, CUDA 블록이 가지는 제약을 고려할 때 얼마나 많은 워프를 활성화할 수 있는지에 따라 병렬화의 성능 지표 중 하나인 워프의 점유율(occupancy)을 계산할 수 있다.

Table 7.은 실험에 사용한 GPU의 구조 및 자원을 설명한다. 레지스터, 공유 메모리, 스레드 항목의

Table 7. NVIDIA GeForce RTX 2080 specifications

Architecture	Turing
Multiprocessors(MP)	48
CUDA cores	3,072
Memory speed	14 Gbps
Register / block	65,536
Register / MP	65,536
Shared memory / block	49,152 bytes
Shared memory / MP	65,536 bytes
Threads / block	1,024
Threads / MP	1,024

기재된 수치는 멀티프로세서 또는 블록 내에서 최대로 사용 가능한 크기를 나타낸다. 위와 같은 제약을 고려하여 워프의 효율성을 최대로 하기 위해서는 커널함수의 실제 메모리 사용량과 그에 따른 성능을 분석하는 것이 필요하다.

본 실험에서 NTRU-hps4096821 파라미터 세트 상에서 Nsight를 통해 실험한 결과 캡슐화 커널과 역캡슐화 커널에서 레지스터를 각각 80개, 82개 사용함을 확인하였다. 이는 다항식 곱셈 연산과 해시함수 과정에서 많은 레지스터를 사용하는 결과로 예상된다. 따라서 캡슐화 커널의 경우 최대 워프 점유율이 75%이며 4개의 블록을 MP에 활성화하는 것이 가능하다. 실제 실험결과 캡슐화 과정을 $N=10,000$번 실행하기 위해 $94.4ms$의 시간이 소요되었다. 역캡슐화 커널의 경우 최대 워프 점유율은 63%이며 4개의 블록을 활성화하는 것이 가능하다. 역캡슐화 과정을 $N=10,000$번 실행하기 위해서는 $87.6ms$의 시간이 소요되었다.

5.2 레이턴시 숨기기

커널함수에서 글로벌 메모리에 접근하는데 발생하는 레이턴시(latency)를 줄이기 위해 커널함수 실행 시 글로벌 메모리로 전달된 키, 암호문 등을 공유 메모리에 저장하여 사용하고 종료 시 다시 글로벌 메모리로 결과를 전달하였다. 이때 저장하는 과정에서 발생하는 레이턴시도 감소시키고자 전체 스레드 수 만큼씩 글로벌 메모리에 접근하도록 하였다. 이 경우 스레드들이 워프 단위로 버스트 섹션(burst section)에 속하는 메모리를 처리하기 때문에 한 스레드가 인접한 위치의 메모리에 연속적으로 접근하는 경우보다 큰 레이턴시 숨기기(latency hiding) 효과를 기대할 수 있다.

5.3 해시함수 커널 분리

본 논문에서 제시한 병렬화 방법에 적합하지 않은 sha3-256 연산은 하나의 스레드가 실행하기 때문에 커널함수 내 블록이 sha3-256 연산을 실행하는 동안 유휴자원의 점유율이 매우 높아지게 된다. 특히, 역캡슐화 커널의 경우 sha3-256 연산이 2회 실행되기 때문에 성능 저하가 클 것으로 보았다. 또한, 역캡슐화 커널에서 워프 효율성의 병목화 현상은 레지스터 사용량에 의해 나타났다. Nsight를 통해 확인한 결과 sha3-256 연산을 제외할 경우, 역캡슐화 커널의 레지스터 사용량은 약 59개로 확인되었

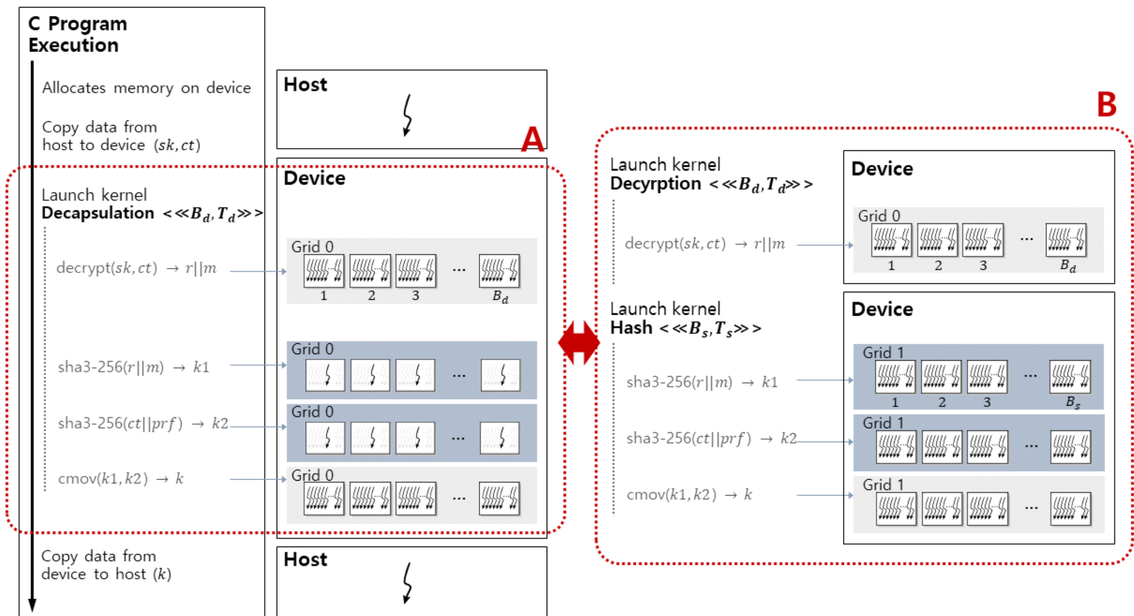

Fig. 11. Host/Device workflow comparison of decapsulation methods: (A) single kernel version and (B) multiple kernel version

다. 이를 통해 sha3-256 연산을 분리하여 실행한다면 성능을 더 높일 수 있을 것으로 예상하였다. 따라서 역캡슐화 커널에서 sha3-256 연산을 다른 커널함수로 분리하는 방법을 제시한다. 즉, 역캡슐화 커널을 복호화 커널과 해시함수 커널로 분리하여 복호화 과정은 Fig.3.에서 기술한 바와 같이 각 블록이 연산을 실행하고 해시함수 과정은 각 스레드가 병렬로 실행하도록 하였다.

Fig.11.의 A는 역캡슐화 과정에서 블록 내의 스레드가 협력하여 실행하는 연산뿐만 아니라 블록 내에서 하나의 스레드가 실행하는 sha3-256 연산까지, 모든 연산을 하나의 커널에서 실행하는 경우이다. 이 경우 sha3-256 연산을 두 번 실행하는 해시함수 과정에서 블록 내의 대부분의 스레드가 활성화되지 않은 상태로 자원을 낭비하게 된다. 결과적으로 한 번에 블록의 개수 B_d만큼 해시함수 과정을 처리하여 공유키를 생성한다. 반면에 Fig.11.의 B와 같이 역캡슐화 과정을 두 커널로 분리하여 실행하는 경우 해시함수 과정을 각 스레드가 병렬로 처리한다. 즉, 제시하는 방법을 통해 해시함수 과정에서의 자원의 낭비를 줄이고 한 번에 $T_d \times B_d$개만큼 해시함수 과정을 처리할 수 있게 된다. 또한, 레지스터 사용량이 많던 연산을 분리하는 것을 통해 복호화 과정의 워프 효율성이 높아졌을 것으로 예상한다. 실제로 해시함수 커널을 분리하여 역캡슐화 과정을 $N=10{,}000$번 실행하였을 때 72.2ms의 시간이 소요되었다. 이는 5.1절과 비교하여 약 1.2배 성능이 향상된 결과이다. 그러나 sha3-256 연산 자체를 병렬화할 수 있다면 성능이 더 향상될 것으로 보인다. 두 커널을 분리하는 데서 발생하는 레이턴시로 인해 성능이 저하되는 측면도 고려해야 하기 때문이다.

5.4 성능측정 결과

본 절에서는 NTRU-hps4096821에 대하여 3장과 4장에서 설명한 병렬화 방법과 5장에서 설명한 고속화 방법을 모두 적용한 결과 전체적인 성능향상을 분석한다. 이때, 각 방법을 적용한 결과는 Fig.2.과 같은 시나리오상에서 최대 성능을 제시하는 것으로, 실제 환경에 적용될 경우 본 논문의 결과보다 지연시간이 생길 수 있다. 성능향상을 위해 적용한 병렬화 및 고속화 방법을 정리하면 다음과 같다.

① KEM 병렬화 (3.2절)
② 연산 고속화 (4.2절)
③ 해시함수 커널 분리 (5.3절)

①은 Fig.3.과 같이 다수의 블록을 사용하여 캡슐화, 역캡슐화 과정을 병렬로 실행하는 것을 의미한다. ②는 이때 하나의 블록 내에서 스레드가 협력하여 연산을 실행하는 방법으로 Table 6.에서 단일 스레드가 실행하는 경우와 비교하여 병렬화 효율성을 제시하였다. ③은 역캡슐화 과정에서 해시함수 과정을 다른 커널로 분리하여 실행하는 기법으로 5.3절에서 해당 방법을 적용할 때 성능향상이 있음을 확인하였다.

Table 8.은 GPU 상에서 이러한 기법들을 적용하였을 때, CPU 상에서의 성능 대비 향상률을 분석한 결과이다. CPU 상의 성능은 NIST에 NTRU의 저자가 제공한 소스를 사용하여 측정한 결과로 앞서 Table 5.에서 제시하였다. Fig.11.의 A와 같이 ①과 ②기법을 적용한 병렬화 결과 캡슐화 커널은 약 49배 성능향상을 보였으며 역캡슐화 커널은 154배 성능향상을 보였다. 역캡슐화 커널의 경우 Fig.11.의 B와 같이 ③의 방법까지 적용하였을 때가 최대 성능임을 확인하였다. 이는 각 사용자가 지연 없이

Table 8. Performance comparison of implementation techniques (N: number of executions)

Implementation / Techniques		$N=10{,}000$			
		encapsulation		decapsulation	
		times (ms)	improvement ratio	times (ms)	improvement ratio
CPU		4704.2	1	11153.3	1
GPU	Single kernel version (①+②)	**94.4**	**×49.8**	87.6	×127.3
	Multi kernel version (①+②+③)	-	-	**72.2**	**×154.5**

캡슐화를 통해 암호문을 전달한다고 가정할 때, 제시한 병렬화 및 고속화 방법을 적용할 경우 서버가 $N=10{,}000$명의 사용자와 약 $80ms$ 이내에 키 교환을 처리할 수 있음을 의미한다.

다음으로 사용자의 수에 따라 서버가 병렬로 처리해야 할 수가 늘어날 때 병렬화 효율성을 확인하기 위해 캡슐화, 역캡슐화를 실행하는 횟수인 N을 증가시키며 실험을 진행하였다. 실험 결과는 Fig.12.-Fig.13.과 같이 NTRU의 캡슐화와 역캡슐화 과정 모두 사용자의 수가 $N=100{,}000$으로 증가할 때까지 CPU 대비 성능향상률이 증가하였다.

일반적으로 N이 충분히 큰 경우, GPU의 최대 성능을 사용하게 된 이후로는 N이 증가하더라도 성능향상률이 계속해서 증가하지 않는 것이 이상적이다. 실험 결과 Fig.12.-Fig.13.과 같이 NTRU의 캡슐화, 역캡슐화 커널은 $N=100{,}000$이후 성능향상률이 증가하지 않고 상수함수의 형태를 보였다. 이 결과로 NTRU의 캡슐화, 역캡슐화 커널은 약 $N=100{,}000$ 이상일 때 GPU의 성능을 최대로 사용하는 것으로 예측할 수 있다.

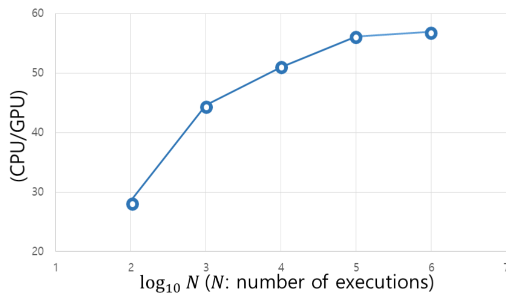

Fig. 12. Improvement ratio of GPU parallel implementation for N encapsulations (log scale)

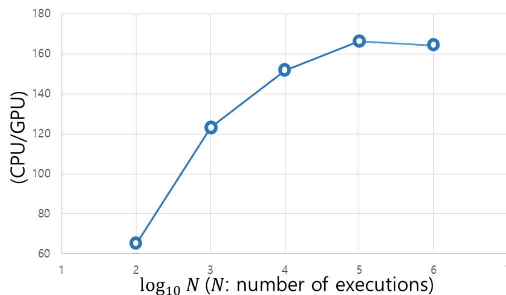

Fig. 13. Improvement ratio of GPU parallel implementation for N decapsulations (log scale)

VI. 결 론

본 논문에서는 TLS1.3 프로토콜에 적합한 파라미터를 가지는 양자내성암호 NTRU를 병렬화 및 고속화하는 것을 통해 다중 사용자 환경에서 사용자와 서버의 키 교환 과정에 적합한 병렬화 시나리오를 제시하였다.

본 논문에서 제시한 방법은 서버에서 NTRU 캡슐화 또는 역캡슐화를 커널함수로 선언하여 병렬로 처리하는 방법으로 병렬화 및 연산 고속화, 해시함수 커널 분리 기법이 적용되었다. 성능향상 결과 사용자의 수가 $N=10{,}000$인 경우 CPU 상의 성능 대비 캡슐화 과정은 최대 약 49배 역캡슐화 과정은 약 154배 성능향상을 달성하였으며 $N=100{,}000$까지 증가시킬 때 더 성능이 향상되었다. 따라서 다수의 사용자 환경에서 서버가 대량의 데이터를 처리해야 할 경우, 본 논문에서 제시한 방법을 활용하여 효과적인 키 교환을 실행 가능할 것으로 예상한다.

References

〔1〕 P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," Proceedings 35th annual symposium on foundations of computer science. IEEE, 1994.

〔2〕 M, Mosca, "Cybersecurity in an era with quantum computers: will we be ready?," IEEE Security & Privacy, 16(5), pp. 38-41, 2018.

〔3〕 CSRC/NIST PQC Project Hompage, "PQC Round 3 Submissions," https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions., last accessed 2021.05.27.

〔4〕 G. Alagic, J.A. Sheriff, and D. Apon, "Status report on the second round of the NIST post-quantum cryptography standardization process," US Department of Commerce, NIST, 2020.

〔5〕 C. Park, Y. Yun, and H, Park, "Implementation of lattice-based post quantum key exchange algorithm," Review of KIISC, 30(3), pp. 11-16, 2020.

〔6〕 K. Jang, M. Sim, and H. Seo, "Design of a lightweight security protocol using post qauntum cryptography," KIPS Trans. Comp. and Comm. Sys. 9(8) (2020): pp. 165-170, 2020.

〔7〕 R. Choi, H. An, and J. Lee, "Comparison of lattice-based key exchange protocols for quantum computing attacks," The Journal of Korean Institute of Communications and Information Sciences. 42(11) (2017): pp. 2200-2207, 2017.

〔8〕 J. Hoffstein, J. Pipher, and J.H. Silverman. "NTRU: A ring-based public key cryptosystem," International Algorithmic Number Theory Symposium. Springer, Berlin, Heidelberg, 1998.

〔9〕 C. Chen, O. Danba, and J. Hoffstein, "NTRU: algorithm specifications and supporting documentation," NIST submissions, Updated Sep. 2020.

〔10〕 J. H. Silverman, "Almost inverses and fast NTRU key creation," report #14, NTRU Cryptosystems Technical Report, 1999.

〔11〕 NVIDIA, CUDA C++ programming guide, PG-02829-001_v11.0, NVIDIA document, Jul. 2020.

〔12〕 E. Crockett, C. Paquin, and D. Stebila, "Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH," IACR Cryptol. ePrint Arch, 2019.

〔13〕 K. Kwiatkowski, "Towards post-quantum TLS," ICMC 2020, 2020.

〔14〕 W. Dai, B. Sunar, and J. Schanck, "NTRU modular lattice signature scheme on CUDA GPUs," International Conference on High Performance Computing & Simulation (HPCS):pp. 501-508. IEEE, 2016.

〈저자소개〉

성 효 은 (Hyoeun Seong) 학생회원
2019년 8월: 국민대학교 수학과 학사
2019년 9월~현재: 국민대학교 일반대학원 금융정보보안학과 석사과정
〈관심분야〉 암호구현 및 분석, 병렬 프로그래밍

김 예 원 (Yewon Kim) 학생회원
2015년 8월: 숙명여자대학교 수학과 학사
2017년 8월: 국민대학교 일반대학원 금융정보보안학과 석사
2017년 9월~현재: 국민대학교 일반대학원 금융정보보안학과 박사과정
〈관심분야〉 암호구현, 난수성 분석 및 평가, 병렬 프로그래밍

염 용 진 (Yongjin Yeom) 종신회원
1991년 2월: 서울대학교 수학과 졸업
1994년 2월: 서울대학교 수학과 석사
1999년 2월: 서울대학교 수학과 박사
2000년 4월~2012년 2월: ETRI 부설연구소 책임연구원/팀장
2012년 3월~현재: 국민대학교 과학기술대학 정보보안암호수학과 정교수
2013년~현재: 국민대학교 BK21+ 안전한 초연결사회를 위한 문제해결형 정보보안 교육연구단 교수
〈관심분야〉 암호구현 및 분석, 보안시스템 평가

강 주 성 (Ju-Sung Kang) 종신회원
1989년 2월: 고려대학교 수학과 졸업
1991년 2월: 고려대학교 일반대학원 수학과 석사
1996년 2월: 고려대학교 일반대학원 수학과 박사
1997년~2004년: 한국전자통신연구원 선임연구원/팀장
2004년 3월~현재: 국민대학교 과학기술대학 정보보안암호수학과 정교수
2013년~현재: 국민대학교 BK21+ 안전한 초연결사회를 위한 문제해결형 정보보안 교육연구단 교수
〈관심분야〉 암호이론, 정보보안 프로토콜, 안전성 분석 및 평가