High Throughput Implementation of Post-quantum Key Encapsulation and Decapsulation on GPU for Internet of Things Applications

Wai-Kong Lee, *Member, IEEE,* and Seong Oun Hwang, *Senior Member, IEEE,*

Abstract—Internet of Things (IoT) sensor nodes are placed ubiquitously to collect information, which is then vulnerable to malicious attacks. For instance, adversaries can perform side channel attack on the sensor nodes to recover the symmetric key for encrypting IoT data. Refreshing the symmetric key frequently can reduce the risk of compromised keys. However, the number of sensor nodes connected to the gateway and cloud server is massive. Refreshed symmetric keys need to be sent to gateway devices and cloud server frequently with a secure key encapsulation mechanism (KEM), which is time-consuming. In this paper, novel and efficient implementation techniques are proposed to accelerate Kyber, a post-quantum KEM, on a Graphics Processing Unit (GPU). Fully parallel implementation of number theoretic transform (NTT) with combined levels is presented, which is 2.65× faster than state-of-the-art result on a GPU. Other proposed techniques include parallel rejection sampling, central binomial distribution with coalesced memory access and parallel fine-grain AES-256. These techniques enable high throughput performance with 162760 encapsulations/second and 107631 decapsulations/second on an RTX2060 GPU. This is also the first fine grain implementation of post-quantum KEM (Kyber) on a GPU, which can be used to offer key encapsulation/decapsulation as a service to reduce the burden on IoT systems.

Index Terms—post-quantum cryptography, key encapsulation mechanism, graphics processing units, Kyber, lattice-based cryptography.

1 Introduction

The Internet of Things (IoT) has developed by leaps and bounds in the past decade. According to an estimation by Statistica [1] in 2018, 23.14 billion devices were already deployed around the world; this number is expected to triple to around 75.44 billion by 2025. With these massively connected IoT sensor nodes, many creative applications like the smart city [2], smart agriculture [3] and Industry 4.0 [4] can be realized. Development of the IoT is expected to continue, opening up many interesting applications in various facets of our lives.

Unlike conventional Internet-based devices, IoT sensor nodes are more vulnerable to malicious attacks, because they can be accessed by the public. For instance, sensor nodes for the smart traffic lights or smart agriculture may be taken over by attackers to perform side channel attacks [5]. Once the symmetric encryption key is stolen, it can be used for various malicious attacks on IoT systems [7]. These attacks may lead to catastrophic effects, such as traffic accidents, unexpected power outage in buildings and leakage of industrial secrets from a smart factory.

On one hand, side channel resistant techniques [6] can be developed to resist malicious attacks. Another way is to frequently refresh the session key used for symmetric encryption, so the effect of a compromised key can be reduced. Random session keys are communicated via gateway device, and this has to be done in a secure manner. This can be achieved through the use of a key encapsulation mechanism (KEM), which is usually built upon conventional public key encryption (e.g. RSA and ECC). In a typical IoT system, the gateway device is usually connected to many sensor nodes. Since the KEM is computationally heavy, the gateway device may not be able to perform all key encapsulation/decapsulation in a timely manner.

Besides the performance issue, a KEM developed based on conventional public key cryptography (PKC) is going to be obsolete in the near future, due to threats from quantum computer attacks [8]. The National Institute of Standards and Technology (NIST) recently organized a global-scale competition [9] to select KEM and signature schemes that are resistant to quantum computer attacks. Currently, there are 15 selected Round-3 candidates in the NIST competition, in which seven of them were developed from the lattice-based hard problems. Kyber, one of the finalists, is among the most suitable KEM for IoT applications, because it has small key sizes and runs fast on various hardware platforms.

In this paper, parallel implementation techniques are proposed to accelerate Kyber in a GPU platform. The proposed implementation techniques can be utilized by the gateway device to handle massive key encapsulation/decapsulation with high throughput. This also paves the way to the possibility of providing key encapsulation/decapsulation as a service (KEDaaS), which follows the business model of security as a service (SECaaS) [11].

• *Wai-Kong Lee and Seong Oun Hwang are with the Department of Computer Engineering, Gachon University, South Korea. E-mail: waikong.lee@gmail.com, sohwang@gachon.ac.kr*

Manuscript received ; revised.

The contribution of this paper are summarized below:

1) The first fine grain implementation of post-quantum KEM (Kyber) in GPU is presented. One of the most time-consuming parts of Kyber is number theoretic transform (NTT). To accelerate NTT, this paper presents a fully parallel implementation that is 1.73× faster than state-of-the-art [10] NTT implementation in GPU. A novel technique is also proposed to combine NTT computation at the first two levels, in order to reduce memory read/write operations, eventually achieving a 2.65× improvement over the technique in [10].
2) Rejection sampling and centered binomial distribution (CBD) in Kyber are also optimized for parallel execution. In this paper, a small modification to the rejection sampling algorithm allows us to perform this operation in parallel. A new technique is also proposed to perform parallel CBD, which is free from non-coalesced memory access pattern.
3) An offloading architecture is presented to demonstrate the practicality of our proposed KEDaaS. This helps to protect IoT systems from malicious attacks without burdening their performance.

2 Background

This section provides an overview of the GPU architecture and presents the selected post-quantum KEM scheme (Kyber). It concludes with a summary of prior work related to this paper.

2.1 Overview of GPU Architecture

A GPU consists of hundreds to thousands of cores; they are grouped into a larger unit called *Streaming Multiprocessor* (SM). For instance, the RTX2060 is a GPU with Turing architecture; there are 34 SMs in the RTX2060, and each SM consists of 64 cores. CUDA is the SDK released by NVIDIA to ease programming of the GPU for general purpose computing. Under the CUDA programming model, multiple threads are organized as a block, and multiple blocks form a grid. Referring to Figure 1, each thread and block can be indexed individually using built-in variables (*threadIdx* and *blockIdx*). To allow efficient instruction scheduling, 32 threads are grouped into one *warps*, in which all threads within the warp execute the same instructions.

2.1.1 Memory Hierachy

The memory hierarchy in the GPU has a deep architecture which is different from the generic CPU. GPU memory can be divided into on-chip memory and off-chip memory with a huge difference in performance. *Global memory* is the off-chip memory (DRAM) in a GPU, which is the largest in size, but the slowest in speed. To optimized the read/write speed in global memory, it is advisable to group the access into contiguous memory locations, so read/write can be performed in a large block. This is termed as "*coalesced memory access*" in CUDA terminology, which is analogous to burst mode in DRAM. *Shared memory* is only accessible by the threads within the same block. It is usually used as the user-managed cache, which is much faster than global

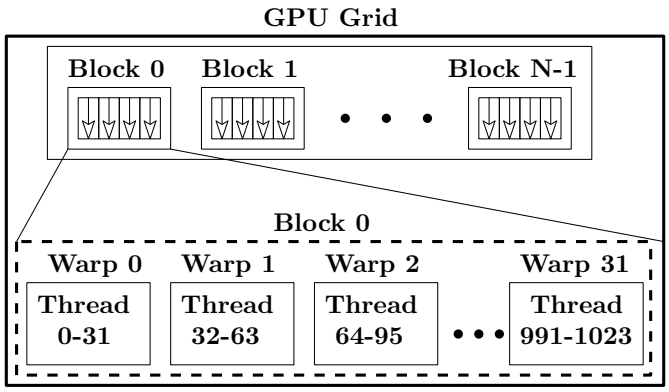

Fig. 1: Relationship between grid, block, warp and thread in CUDA

memory. The fastest memory in a GPU is the *registers*, but registers have a very limited size (e.g. 64K words per SM for the RTX2060), and they are only accessible locally by each thread.

2.2 Overview of Kyber: Post-quantum Key Encapsulation Mechanism

Kyber is a lattice-based KEM based on the hardness of solving the learning-with-errors problem in module lattices (MLWE [27]). The scheme has compact keys and excellent speed, making it a strong candidate in the post-quantum world. In July 2020, Kyber was selected as one of the seven finalists in the NIST standardization competition [9]. The Kyber KEM [28] consists of three algorithms (Kyber.KEM.KeyGen, Kyber.KEM.Encap and Kyber.KEM.Decap), which are constructed based on the public key encryption (PKE) scheme (Kyber.PKE). These algorithms are explained as follow.

Algorithm 1: Kyber.KEM.KeyGen()

Output: Public key $pk \in B^{12\times k\times n/8+32}$
Output: Secret key $sk \in B^{24\times k\times n/8+96}$
$z \leftarrow B^{32}$
(pk, sk'):= Kyber.PKE.KeyGen()
sk:=$(sk'||pk||H(pk)||z)$
return (pk, sk)

B represents an unsigned 8-bit integer (bytes), and B^k refers to a byte array of length k, while B^* means a byte array with an arbitrary length. Algorithm 1 generates a pair of public and private key (pk, sk) through the key generation algorithm (Kyber.PKE.KeyGen) for Kyber public key encryption.

The encapsulation process (Algorithm 3) starts by hashing a random array m and concatenating it with the hash of public key pk. The results are hashed by another hash function G to generate $\hat{K}$ and r. The value r is encrypted by Kyber.PKE.Enc to output a ciphertext c; it is then hashed and concatenated with $\hat{K}$ and passed to a key derivation function (KDF) to generate K. Finally, the algorithm ends the encapsulation by producing ciphertext c and shared key K.

Algorithm 2: Kyber.KEM.Encap()

Input: Public key $pk \in B^{12\times k\times n/8+32}$
Output: Ciphertext $c \in B^{d_u\times k\times n/8+d_u\times n/8}$
Output: Shared key $K \in B^*$
1 $m \leftarrow B^{32}$
2 $m \leftarrow H(m)$
3 $(\hat{K}, r)$:=$G(m||H(pk))$
4 c := Kyber.PKE.Enc(pk, m, r)
5 K := KDF$(\hat{K}||H(c))$
6 **return** (c, K)

Algorithm 3: Kyber.KEM.Decap()

Input: Ciphertext $c \in B^{d_u\times k\times n/8+d_v\times n/8}$
Input: Secret key $sk \in B^{24\times k\times n/8+96}$
Output: Shared key $K \in B^*$
1 pk:=$sk + 12 \times k \times n/8$
2 h:=$sk + 24 \times k \times n/8 + 32 \in B^{32}$
3 z:=$sk + 24 \times k \times n/8 + 64$
4 m':= Kyber.PKE.Dec$(s, (u, v))$
5 $(\hat{K}', r')$:=$G(m'||h)$
6 c':= Kyber.PKE.Enc(pk, m', r')
7 **if** $c = c'$ **then**
8 **return** K:=KDF$(\hat{K}'||H(c))$
9 **else**
10 **return** K:=KDF$(z||H(c))$
11 **return** K

Referring to Algorithm 3, the decapsulation process first decrypts the ciphertext through Kyber.PKE.Dec to recover $m', \hat{K}'$ and r'. Then, with the same public key pk, Kyber.PKE.Enc encrypts m' and r' to generate the ciphertext c'. The received ciphertext c is compared against c' to recover shared key K.

Algorithm 4 describes the Kyber PKE key generation process in detail. Random sequences are first generate through an extendable output function (XOF). Matrix $\hat{A}$ is then generated and stored in NTT form (lines 8-10) through the rejection sampling function (Parse). Noise is sampled by using the centered binomial distribution (CBD) function and converted to the NTT domain (lines 11-12). Matrix $\hat{A}$ is multiplied with the noise vector $\hat{s}$ and adds noise vector $\hat{e}$ (line 13). Finally, the public and private key are produced by encoding $\hat{s}$ and $\hat{t}$.

Kyber PKE encryption is described in Algorithm 5. Similar to Kyber.PKE.KeyGen(), a matrix $\hat{A}^T$ is generated and stored in NTT form (lines 4-6). Note that $\hat{A}^T$ is essentially the transpose of $\hat{A}$, and they should use the same XOF function. Noises $(r, e_1 and e_2)$ are sampled by using the CBD function and converted to the NTT domain (lines 7-12). Note that e_1 and e_2 are assumed to already be in the NTT domain, so no conversion is required. Noise vector $\hat{r}$ is multiplied with matrix $\hat{A}^T$ and adds noise vector e_1. Then, the decoded public key $(\hat{t})$ is multiplied with $\hat{r}$ and adds e_2 and decompressed message m (line 14). Finally, ciphertext c_1 and c_2 are generated by encoding the compressed u and v.

Algorithm 4: Kyber.PKE.KeyGen() key generation

Output: Secret key $sk \in B^{12\times k\times n/8}$
Output: Public key $pk \in B^{12\times k\times n/8+32}$
1 $d \leftarrow B^{32}$
2 (ρ, σ):=G(d)
3 N:=0
`/* Generate Matrix` $\hat{A} \in R_q^{k\times k}$ `in NTT domain */`
4 **for** *i from 0 to k-1* **do**
5 **for** *j from 0 to k-1* **do**
6 $\hat{A}$[i][j]:=Parse(XOF(ρ, j, i))
7 **for** *i from 0 to k-1* **do**
`/* Sample` $s \in R_q^k$ `and` $e \in R_q^k$ `from` B_q `*/`
8 s[i] := CBD$_\eta$(PRF(ρ, N))
9 e[i] := CBD$_\eta$(PRF(ρ, N))
10 N:= N+1
11 $\hat{s}$:= NTT(s);
12 $\hat{e}$:= NTT(e);
13 $\hat{t} := \hat{A} \circ \hat{s} + \hat{e}$;
14 pk:= (Encode$_{12}(\hat{t}\ mod^+ q)||\rho$)
15 sk:= Encode$_{12}(\hat{s}\ mod^+ q)$)
16 **return** (pk, sk)

The public key decryption process is relatively simple. It first decompress the decoded ciphertext (lines 1 and 2) into u and v. Then, u is transformed to the NTT domain and multiplied with the transpose of the decoded secret key *sk*. The results are converted back to normal form (NTT^{-1}) and subtracted from v. Finally, message m is recovered by encoding the compressed results.

Kyber employs rejection sampling on uniformly distributed random bytes to obtain the samples in R_q. This process is described in Algorithm 7. Noise (used in Algorithm 4 and 5) in Kyber is sampled from a CBD B_η, where $\eta = 2$. This is described in Algorithm 8.

Note that Kyber proposed two sets of symmetric primitives to be used according to the implementation environment. Kyber originally utilized Keccak for all symmetric primitives, but it also provide the "90s" variant that uses AES-256. In this paper, we adopt the "90s" variant of Kyber; the symmetric primitives employed in our implementation are described in Table 1. XOF and a pseudorandom function (PRF) are implemented through AES with a 256-bit key. AES-CTR was selected for this paper because it allows parallel implementation in GPU. XOF and PRF can also be implemented through SHAKE-128 and SHAKE-256, but these two algorithms are serial and not suitable for GPU implementation. Two hash functions, *H* and *G*, are implemnted through SHA-256 and SHA-512 respectively. The key derivation function (KDF) uses SHAKE-256 to derive the shared key.

Table 2 shows the parameter sets in Kyber that covers 128-bit, 192-bit and 256-bit security levels. Note that Kyber is designed to allow easy implemention for different security levels by adjusting only k and η; the polynomial degree (n) and modulus (q) remain the same across all security levels.

Algorithm 5: Kyber.PKE.Enc() public key encryption

Input: Message $m \in B^{32}$
Input: Random coins $r \in B^{32}$
Input: Public key $pk \in B^{12\times k\times n/8+32}$
Output: Ciphertext $c \in B^{d_u\times k\times n/8+d_v\times n/8}$

N:=0
$\hat{t}$:=$Decode_{12}(pk)$)
$\rho := pk + 12 \times k \times n/8$
/* Generate Matrix $\hat{A}^T \in R_q^{k\times k}$ in NTT domain */
for *i from 0 to k-1* **do**
  for *j from 0 to k-1* **do**
    $\hat{A}^T$**[i][j]**:=Parse(XOF(ρ, i, j))
for *i from 0 to k-1* **do**
  /* Sample $r \in R_q^k$ and $e_1 \in R_q^k$ from B_η */
  r[i] := CBD$_\eta$(PRF(r, N))
  e$_1$[i] := CBD$_\eta$(PRF(r, N))
  N:= N+1
/* Sample $e_2 \in R_q^k$ from B_η */
e_2 := CBD$_\eta$(PRF(r, N))
$\hat{r}$:= NTT(r);
u := NTT^{-1}($\hat{A}^T \circ \hat{r}$) + e_1;
v := NTT^{-1}($\hat{t}^T \circ \hat{r}$) + e_2 + Decompress$_q$(Decode$_1$(m), 1);
c_1:= Encode$_{d_u}$(Compress$_q$(u, d_u))
c_2:= Encode$_{d_v}$(Compress$_q$(v, d_v))
return c = (c_1, c_2)

Algorithm 6: Kyber.PKE.Dec() public key decryption

Input: Secret key $sk \in B^{12\times k\times n/8}$
Input: Ciphertext $c \in B^{d_u\times k\times n/8+d_v\times n/8}$
Output: Message $m \in B^{32}$

u := Decompress$_q$(Decode$_{d_u}$(c), d_u)
v := Decompress$_q$(Decode$_{d_v}$($c + d_u \times k \times n/8$), d_v)
$\hat{s}$:= Decode$_{12}$(sk)
m := Encode$_1$(Compress$_q$($v-$ NTT^{-1}($\hat{s}^T \circ$ NTT(u)),1))
return m

Algorithm 7: Parse (rejection sampling): $B^* \rightarrow R_q^n$

Input: Byte stream $B = b_0, b_1, ... \in B^*$
Output: NTT representation $\hat{a} \in R_q$ of $a \in R_q$

$i := 0$
$j := 0$
while $j < n$ **do**
  d : = b_i + 256 $\times b_{i+1}$
  if $d < 19 \times q$ **then**
    $\hat{a_j}$:= d
    $j := j + 1$
  $i := i + 2$
return $\hat{a_0} + \hat{a_1}X + ... + \hat{a}_{n-1}X^{n-1}$

Algorithm 8: CBD$_\eta$: $B^{64\eta} \rightarrow R_q$

Input: Byte array $B = b_0, b_1, ..., b_{64\eta-1} \in B^{64\eta}$
Output: Polynomial $f \in R_q$

($\beta_0, ..., \beta_{512\eta-1}$) := BytesToBits($B$)
for *i from 0 to 255* **do**
  $a := \sum_{j=0}^{\eta-1} \beta_{2\times i\times\eta+j}$
  $b := \sum_{j=0}^{\eta-1} \beta_{2\times i\times\eta+\eta+j}$
  $f_i := a - b$
return $f_0 + f_1X + ... + f_{255}X^{255}$

2.3 Related Work

In view of the increasing security breaches, SECaaS has recently become an emerging business model in service computing. For instance, Tian et al. [13] presented an interesting work to securely offload RSA key generation. Chang et al. [14] had also proposed a similar architecture for IoT applications, wherein RSA signatures generation is offloaded to a gateway device equipped with GPU. The work from Dai et al. [15] is among some early research that optimize implementation of post-quantum signature generation in a GPU platform. They implemented the NTRU-MLS [16] signature scheme with a very high acceptance rate for rejection sampling, which was $47\times$ faster than a CPU implementation. Another notable example is signature as a service (SIGaaS) presented by Sun et al. [12], wherein multiple GPUs are utilized to perform high throughput signature generations/verifications. This work implemented another post-quantum signature scheme: SPHINCS [17]. Our work follows the similar rationale, in which the GPU is utilized to perform high throughput key encapsulation/decapsulation to support IoT systems.

One of the most time-consuming operations in Kyber is NTT, where its implementation in a GPU has been studied in the past. Emmart and Weem [18] implemented the Strassen multiplication algorithm for integer multiplication utilizing an NTT algorithm. Their implementation showed very good performance for large operand sizes up to 16320K-bit. Later on, Wang et al. [19] presented another NTT implementation to accelerate fully homomorphic encryption on a GPU, based on techniques developed by Emmart and Weems [18]. However, the NTT implementation techniques presented in these two prior studies are not directly applicable to the Kyber KEM, due to the differences in NTT size, the prime field and other parameters.

Other existing studies that implemented NTT for lattice-based cryptosystems, are more closely related to our work. Akleylek and Tok [20] presented a coarse grain implementation of NTT that high latency compared to our approach. Another similar study compared the implementation of serial and parallel NTT against the pre-built library (cuFFT) in a GPU [21]. cuFFT-based NTT was found to have better performance for large-degree polynomials (> 2048). However, the work in [21] implemented sparse polynomial multiplication, which is completely different from the NTT in Kyber. Recently, Lee et al. [10] proposed a strategy to optimize parallel implementation of NTT in GPU, in which there is at least 32 parallel threads to be executed. This

TABLE 1: Symmetric primitives for Kyber implementation

Function name	Symmetric primitives
XOF	AES-256 in CTR mode where ρ is used as the key and $i\|\|j$ is zero-padded to a 12-byte nonce.
H	SHA-256
G	SHA-512
PRF	AES-256 in CTR mode where s is used as the key and b is zero-padded to a 12-byte nonce.
KDF	SHAKE-256

TABLE 2: Parameter sets for Kyber

	Security Level	n	k	q	$\eta\ (d_u, d_v)$	δ
Kyber512	128-bit	256	2	3329	(10, 3)	2^{-178}
Kyber768	192-bit	256	3	3329	(10, 4)	2^{-164}
Kyber1024	256-bit	256	4	3329	(11, 5)	2^{-174}

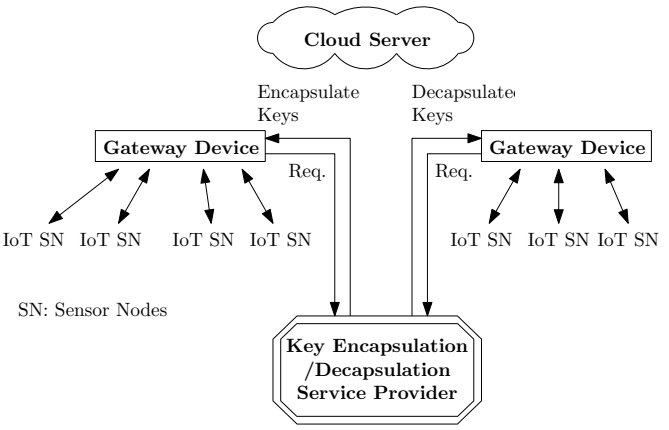

Fig. 2: The proposed KEDaaS architecture for IoT Systems

can avoid warp divergence in the GPU, which is a key performance bottleneck. However, such a technique is not optimized for polynomial with degree $n < 1024$. Moreover, this NTT implemention is not fully parallel; the parallelism varies across different NTT levels. In our paper, we improve upon their work by proposing a fully parallel NTT implementation, that has better performance. Another relevant work was reported by Gupta et al. [22], where they present an coarse grain parallel implementation of Kyber on various GPU. Our approach is different from theirs as we focus on fine grain approach that achieves a balanced throughput and latency performance.

Efficient implementation of Kyber on various hardware platforms also attracted some attention in the research community. Botros et al. [23] presented an implementation of Kyber in Cortex-M4 microcontroller. Their work made use of vectorized DSP instructions to achieve 18% faster software speed with reduced RAM consumption. Chen et al. [24] presented an efficient implementation on an FPGA platform, wherein a bottleneck to memory access was improved by using a dual-column sequential scheme. They also proposed a pipeline architecture for better performance. On the other hand, Sapphire [25] is a notable work that implements a few post-quantum cryptosystems (including Kyber) into a chip under a TSMC 40nm low-power CMOS process. However, no prior work that implemented post-quantum KEM schemes into a massively parallel architecture like GPU. This motivated us to research parallel implementation techniques for accelerating Kyber in a GPU.

3 Key Encapsulation/Decapsulation as a Service for IoT Systems

This section describes the proposed KEDaaS architecture to be used in IoT systems. The symmetric encryption keys can be refreshed frequently to protect the IoT communication, wherein the risks of compromised keys is reduced. This can be carried out in two ways:

1) The IoT sensor nodes generate new session keys for symmetric encryption and send them to the gateway device via KEM. Typically, the symmetric key is refreshed in every communication session, wherein pseudorandom number generator (PRNG) or KDF can be employed to generate the symmetric key.
2) The gateway device generates many new session keys and send them to each sensor node for update, via KEM. In such a scenario, the gateway device can decide the time interval for refreshing the symmetric keys. In other words, the symmetric key can be refreshed every communication session, every hour, or every day, depending on the required level of security.

The first way relies on the IoT sensor nodes to generate a new symmetric key from time to time (e.g. every new session) and to update the gateway device about this change. This requires the gateway device to handle many key decapsulations in real time. On the other hand, the second way requires the gateway device to generate and encapsulate many keys, so they can be sent to each sensor node in a timely manner. Both ways require the gateway device to compute a key encapsulation/decapsulation algorithm at high throughput, which is difficult even for a high-end workstation. Moreover, the gateway device also needs to perform other computational tasks (e.g. edge computing [26] and data aggregation), so it may not have sufficient resources to handle massive key encapsulations/decapsulations.

To reduce the burden of the gateway device, the massive key encapsulations/decapsulations can be offloaded to a GPU inside the gateway device. However, this requires the gateway device to host a GPU, which can be costly to operate an IoT system with many gateway device. Hence, we proposed the KEDaaS architecture to resolve this issue, in which encapsulations/decapsulations are offloaded to a third-party service provider. This allows the IoT system to operate at a lower hardware cost while ensuring cybersecurity without degrading performance speed.

Figure 2 shows the architecture of the proposed KEDaaS. The IoT system subscribed to the KEDaaS can request the service provider to generate symmetric keys and encapsulate them. The encapsulated keys are then sent to the IoT gateways and are distributed to their respective IoT sensor nodes. Since the symmetric keys can be generated prior to a request from IoT systems, KEDaaS can also help to reduce latency in the key encapsulation process. On the

other hand, an IoT system can send the encapsulated keys from IoT sensor nodes to the service provider and request for a key decapsulation service. The decapsulated keys are communicated to the gateways via secure channels.

High throughput key encapsulation and decapsulation with moderate latency is the key enabling technology needed in order to provide KEDaaS to the IoT systems. In this paper, several techniques are proposed to compute high throughput Kyber KEM on a GPU platform. The proposed techniques and implementation details are described in the following section.

4 Parallel Implementation of Kyber on GPU Platform

TABLE 3: Timing performance breakdown of Kyber.KEM.Encap for Kyber512 90's variant

Operation	Cycles	(%)
Kyber.PKE.Enc		
gen_at	76820	47.4
poly_getnoise (executed five times)	24290	15.0
NTT	7356	4.5
INVNTT (executed two times)	15190	9.4
Miscellaneous	22089	13.6
Hash and KDF	16217	10.1
Total	161962	100

This section presents the techniques proposed to accelerate parallel implementation of Kyber on a GPU platform. In particular, we only focus on accelerating Kyber.PKE.Enc and Kyber.PKE.Dec on the GPU, while the other operations remain to be executed on a CPU. Note that in Kyber.KEM.Encap, Kyber.PKE.Enc is the most time-consuming task, and that in Kyber.KEM.Decap, one also needs to execute Kyber.PKE.Enc (see Algorithm 3, line 6) to decapsulate the shared key. This implies that accelerating Kyber.PKE.Enc can greatly improve the throughput of both key encapsulation and decapsulation.

Table 3 shows the breakdown of all important operations in Kyber.KEM.Encap (Algorithm 2) that affects timing performance on an Intel i9-9700F processor. We can observe that the most time-consuming (47.4%) operation comes from gen_at, which corresponds to lines 4-6 in Algorithm 5. The second most time-consuming operations, poly_noise combines PRF and CBD steps (lines 7-10) in Algorithm 5, which are executed five times for Kyber512. Next, NTT and inverse NTT are used to accelerate the multiplications of polynomials; they consumes 13.9% of the total encapsulation time. The remaining miscellaneous operations (including Encode, Decode, Compress, Decompress and pointwise multiplication)in Kyber.PKE.Enc consumes 13.6% of the total time. Finally, initialization, the generation of the shared key and hashing (lines 1, 2, 3 and 5 in Algorithm 2) consumes only 10.1% of total time. This implies that gen_at, poly_noise, NTT and inverse NTT (INVNTT) are the key operations that affect the performance. Based on this observation, our GPU implementation focuses on optimizing these four operations to achieve high throughput key encapsulation/decapsulation. Besides, the miscellaneous operations in Kyber.PKE.Enc are also implemented in GPU to avoid constantly transferring data between the CPU and GPU. The details of our proposed optimization techniques are presented in the subsequent subsections.

4.1 Exploiting Parallelism for Implementing KEM

There are two common methods to implementing an algorithm in a GPU: coarse-grain and fine-grain parallelism. To implement KEM, coarse grain method computes one KEM per thread; this is essentially a serial implementation. On the other hand, fine-grain method utilizes multiple threads to compute one KEM. Coarse-grain method can provide high throughput, since there is no data synchronization between each thread. However such an approach has high latency, because each thread computes one KEM completely in serial; this is not desirable in IoT applications that demand a decent response time. In this paper, we employed fine-grain parallelism to implement Kyber, wherein many blocks are launched in parallel and each block (with multiple threads) computes one KEM.

4.2 Number Theoretic Transform

Algorithm 9 shows the pseudocode of a typical in-place implementation of NTT, wherein the output is written into the input array to save memory. The twiddle factors (zeta) are pre-computed and stored in a lookup table (zeta_tb) to improve the speed. The function **fqmul** refers to multiplication followed by a modular reduction with a prime number q, where $q = 3329$ for Kyber.

Algorithm 9: Pseudocode: In-place NTT in Kyber

```
Input: Polynomial r with degree of n
Output: Polynomial r in NTT domain
1 k = 1;
2 for len=128; len≥ 2; len=len/2 do
3     for start=0; start< 256; start=j+len do
4         zeta = zeta_tb[k++];
5         for j=start; j< start + len; j=j+1 do
6             t = fqmul(zeta, r[j + len]);
7             r[j + len] = r[j] - t;
8             r[j] = r[j] + t;
```

In this paper, we proposed three different techniques to improve the performance of NTT implementation. The first technique allows a fully parallel implementation in GPU, which can fully exploit the massively parallel architecture in GPU. This is an improvement over the previous implementation by Lee et al. [10]. The second technique focuses on the intelligent use of shared memory to reduce the access to a slow global memory. Lastly, we combined the last two levels in Kyber NTT in order to reuse the computed data stored in registers, in turn reducing the read/write into the slower shared memory. All these techniques are described in the subsequent subsections.

4.2.1 Fully Parallel NTT Implementation

Referring to Algorithm 9, there are three **For** loops in NTT, which can be parallelized for GPU implementation. Lee et al. [10] proposed a strategy to parallelized the two inner **For** loops (lines 3 and 5) in the GPU. Under such a strategy,

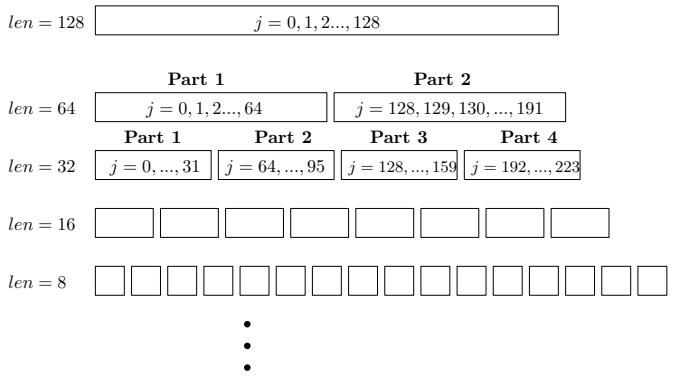

Fig. 3: Data processing patterns of the fully parallel NTT at various levels in Kyber

the parallel loops are interchanged at the middle point of the decomposition level. For instance, a polynomial with $n = 1024$, NTT takes 10 levels to complete; Lee et al. [10] proposed to interchanging the parallel loop at level 5 to ensure there is at least 32 (2^5) parallel threads to avoid warp divergence. However, this does not work for Kyber with $n = 256$, because the interchange point at level 4 only contains 16 (2^4) threads, so warp divergence still happens. To mitigate this issue, we proposed to fully parallelize both **For** loops with an appropriate indexing method in order to achieve higher speed for NTT implementation.

Figure 3 illustrates the data processing patterns of the fully parallel NTT proposed in this paper. At level $len = 128$, index j range from 1-128, which is trivial to parallelize it in GPU by launching 128 threads, with each thread computing a pair of butterfly operation (lines 7 and 8). However, at level $len = 64$, index j is divided into two parts: part 1 (0-63) and part 2 (128-191), wherein Lee et al. [10] only managed to parallelize part 1. Close observation of Algorithm 9 reveals that part 2 does not have any data dependency on part 1. Hence, both parts can be executed in parallel to fully utilize the parallelism in the GPU. Similarly, level $len = 32$ consists of four parts, but there is no data dependency between them. This pattern is also found in the subsequent levels, which indicates that we can actually launch 128 threads and compute all the levels in parallel. This can be achieved through transforming the index j into a GPU-friendly manner, so that each thread can process independent data without any race condition. The proposed fully parallel technique is described in Algorithm 10.

Note that our proposed fully parallel NTT implementation is different from the AVX2 implementation [28]. The AVX2 offers 256-bit registers, which allow at most 16 NTT points (each NTT point is 16-bit) to be computed in parallel; it takes a few iterations to complete one level of NTT computation. However, our proposed implementation allows the entire NTT level with 128 threads to be computed in parallel, which greatly improves the parallelism.

4.2.2 Cached Data in Constant and Shared Memory

The twiddle factors are pre-computed and stored in constant memory (*zeta_tb*) for fast access. The polynomial in Kyber is degree $n = 256$, which is small enough to fit into the shared memory. Hence, we proposed storing the entire polynomial into shared memory to allow faster access. In particular, polynomial r in Algorithm 10 is first loaded from global memory into shared memory, followed by the core NTT operations (lines 2-8). Once the NTT computation completes, the resulting data are copied from shared memory back to global memory.

Algorithm 10: Pseudocode: Fully parallel in-place NTT in Kyber

Input: Polynomial r with degree $n = 256$
Output: Polynomial r in the NTT domain

```
1 level = 1;
  /* 128 threads are laucneded in
     parallel, where tid refers to the ID
     of each thread. */
2 for len=128; len≥ 2; len=len/2 do
      /* Compute each level in parallel */
3     zeta = zeta_tb[level + ⌊tid/len⌋];
4     j = ⌊tid/len⌋ * len + tid;
5     t = fqmul(zeta, r[j + len]);
6     r[j + len] = r[j] - t;
7     r[j] = r[j] + t;
8     level = level × 2;
```

4.2.3 Combining Levels in NTT

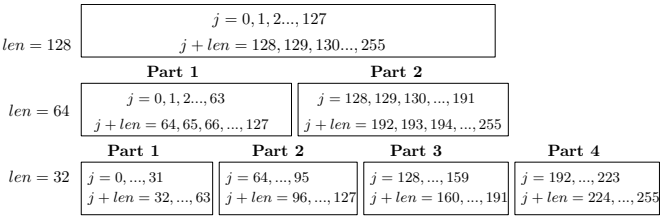

Fig. 4: Data processing patterns at various NTT levels for level combination

Figure 4 shows the detailed data processing pattern in NTT, where index j and $j + len$ correspond to the butterfly operations in lines 5-7 in Algorithm 10. A closer look at this pattern reveals that the data processed in one level are used immediately in the next level. This gives us a chance to combine some levels, so the data are not written back to the polynomial array frequently. For instance, at level $len = 128$, the data written to index 0-127 are consumed by part 1 at level $len = 64$, while the data written to index 128-255 are consumed by part 2 at level $len = 64$. Thus, instead of storing these data back to the polynomial array in shared memory, we proposed caching them in a register to reduce accessing shared memory. The proposed technique that combines level $len = 128$ and $len = 64$ in Kyber NTT is described in Algorithm 11.

Only 64 threads are launched to compute the combined NTT in Algorithm 11. The intermediate data computed by the butterfly operations (lines 7, 8, 10 and 11) at level $len = 128$ are stored in registers $g1 - g4$. In the next level $len = 64$, the results in $g1 - g4$ are consumed. With this technique, we can combine these two levels, cache the intermediate data in registers and reduce accessing shared memory.

Algorithm 11: Pseudocode: Combining the first two levels in Kyber NTT

Input: Polynomial r with degree $n = 256$
Output: Polynomial r in the NTT domain

```
1  level = 1;
   /* 64 threads are laucned in parallel,
      where tid refers to the ID of each
      thread. */
2  len = 128;
3  zeta = zeta_tb[level + ⌊tid/len⌋];
4  j1 = ⌊tid/len⌋ * len + tid;
5  j2 = ⌊tid/len⌋ * len + tid + 64;
6  t = fqmul(zeta, r[j1 + len]);
7  g1 = r[j1] - t;                    // r[j1 + len]
8  g2 = r[j1] + t;                          // r[j1]
9  t = fqmul(zeta, r[j2 + len]);
10 g3 = r[j2] - t;                    // r[j2 + len]
11 g4 = r[j2] + t;                          // r[j2]
12 level = level × 2;
13 len = 64;
14 zeta = zeta_tb[level + ⌊tid/len⌋];
15 j1 = ⌊tid/len⌋ * len + tid;
16 j2 = ⌊tid/len⌋ * len + tid + 128;
17 t = fqmul(zeta, g4);
18 r[j1 + len] = g2 - t;               // r[j1 + len]
19 r[j1] = g2 + t;                          // r[j1]
20 zeta = zeta_tb[level + ⌊(tid + 64)/len⌋];
21 t = fqmul(zeta, g3)]);
22 r[j2 + len] = g1 - t;
23 r[j2] = g1 + t;
24 level = level × 2;
25 for len=32; len≥ 2; len=len/2 do
       /* Compute each level in parallel */
26     zeta = zeta_tb[level + ⌊tid/len⌋];
27     j = ⌊tid/len⌋ * len + tid;
28     t = fqmul(zeta, r[j + len]);
29     r[j + len] = r[j] - t;
30     r[j] = r[j] + t;
31     level = level × 2;
```

TABLE 4: Intermediate data to be stored in registers

Register name	Stored by level $len = 64$ (index)
g1	64-127
g2	0-63
g3	192-255
g4	128-291

However, this technique is not applicable to levels $len \leq 32$. For instance, if the proposed technique is applied to level $len = 32$, then level $len = 64$ should store the intermediate data into the indices as described in Table 4. Since there are 64 threads launched in parallel, level $len = 32$ should access index $j = 0, ..., 31$ and $j = 64, ..., 95$ in parallel (see Figure 4). However, the intermediate data (from level $len = 64$) $j = 0, ..., 31$ are stored in *g2*, while the data for $j = 64, ..., 95$ are stored in *g1*. This implies that the first 32 threads and the next 32 threads have to read different registers, and thus require an if/else statement to control execution. Note that this is not optimized for GPU execution due to the extra overhead to perform the predication in if/else control. Furthermore, at level $len \leq 16$, the if/else control is performed at a level smaller than 32 threads, which creates warp divergence and seriously degrades the performance. For these reasons, we do not combine levels $len \leq 32$; they are executed as ordinary NTT (see lines 25-31, Algorithm 10).

4.2.4 Micro-benchmark of NTT Implementation

To evaluate the effectiveness of each proposed NTT techniques, the following implementations were performed.

1) NTT-1: From Lee et al. [10]
2) NTT-2: Fully parallel NTT
3) NTT-3: Fully parallel NTT + cached data
4) NTT-4: Fully parallel NTT + cached data + combined first two levels

NTT-1 is the implementation that follow the techniques proposed by Lee et al. [10]. Experiments NTT-2, NTT-3 and NTT-4 are performed by launching 16384 blocks with 128 or 64 threads each. These experiments are repeated for 100 times, where the average results are recorded in Table 5.

TABLE 5: Timing performance from the NTT implementation techniques

Techniques	Blocks	Parallel Threads	Time (ns)
NTT-1	16384	16-128	45
NTT-2	16384	128	26
NTT-3	16384	128	21
NTT-4	16384	64	17

With the proposed fully parallel NTT (NTT-2), we achieved a speed $1.73\times$ faster than Lee et al. [10]. After caching the twiddle factors and polynomial in constant memory and shared memory respectively, the results (21ns) were improved by another 23.81%. NTT-4 shows the best result (17ns) by combining all the proposed techniques, which is $2.65\times$ faster than Lee et al. [10]. The results from NTT-4 was 23.53% faster than NTT-3, which implies that combining level 7 and 6 effectively improved performance. Note that NTT-4 has smaller parallelism compared to NTT-3, but the performance gain is high enough to offset this shortcoming. The inverse NTT is very similar to NTT except that the twiddle factors are replaced with inverse twiddle factors. Hence, the three proposed techniques can be applied to INVNTT as well. For brevity, we do not describe the implementation of INVNTT in this paper.

4.3 Rejection Sampling

Rejection sampling is used together with the XOF function in the gen_at operation to generate random matrices. Referring to Algorithm 7, the rejection sampling checks input byte stream B element by element serially (line 3). However, only the samples that falls within range (line 5) are being used; other samples are rejected. For this reason, index j is only increased if the samples are valid. This inconsistent indexing poses a challenge to implementing rejection sampling in parallel. To mitigate this problem, we proposed a simple parallel rejection sampling algorithm described in Algorithm 12.

Algorithm 12: Pseudocode: Parallel parse (rejection sampling): $B^* \rightarrow R_q^n$

Input: Byte streams $B1 = b1_0, b1_1, ... \in B^*$ and $B2 = b2_0, b2_1, ... \in B^*$
Output: NTT representation $\hat{a} \in R_q$ of $a \in R_q$
`/* n=256 threads are launched in parallel, where tid refers to the ID of each thread. */`
1 $d := b1_{tid\times 2} + 256 \times b1_{tid\times 2+1}$
2 **if** $d < 19 \times q$ **then**
3 $\quad \hat{a_{tid}} :=$ d
4 $d := b2_{tid\times 2} + 256 \times b2_{tid\times 2+1}$
5 **if** $d < 19 \times q$ **then**
6 $\quad$ **if** $\hat{a_{tid}} = 0$ **then**
7 $\quad\quad \hat{a_{tid}} :=$ d`// Only replaces the zero elements.`
8 **return** $\hat{a_0} + \hat{a_1}X + ... + \hat{a}_{n-1}X^{n-1}$

Two byte streams $B1$ and $B2$ are generated in advance. For the implementation in Kyber, 256 threads are launched in parallel to perform rejection sampling. Each thread constructs the samples (line 1) from $B1$ and checks the range (line 2) in parallel; only the samples that fall within range are accepted (line 3). This process is repeated, where more samples are constructed from $B2$ (line 4), but only locations without any valid samples ($\hat{a_{tid}} = 0$) are updated (line 7).

4.4 Noise Generation: Central Binomial Distribution

Algorithm 13: Pseudocode: CBD implemented in Kyber

Input: Byte array $B = b_0, b_1, ..., b_{64\eta-1} \in B^{64\eta}$
Output: Polynomial $f \in R_q$
1 **for** *i=0; i*$< n/8$*; i++* **do**
2 $\quad t =$ BytesTo32Bits($B_{4\times i}$)`// Load four bytes`
3 $\quad d = t \wedge$ `0x55555555`;
4 $\quad d+ = (t \gg 1) \wedge$ `0x55555555`;
5 $\quad$ **for** *j=0; j*< 8*; j++* **do**
6 $\quad\quad a = (d \gg 4 \times j) \wedge$ `0x3`;
7 $\quad\quad b = (d \gg 4 \times j + 2) \wedge$ `0x3`;
8 $\quad\quad f_{8\times i+j} := a - b$
9 **return** $f_0 + f_1X + ... + f_{255}X^{255}$

The poly_noise operation first generates random bytes through AES-256-CTR, then produce a random polynomial through CBD (Algorithm 8). The CBD operation implemented in Kyber is detailed in Algorithm 13. This CBD operation can be implemented in GPU by parallelizing the i loop (line 1), which is also described in Algorithm 14.

A closer look at Algorithm 14 reveals that this naïve implementation causes two serious performance issues. The access to the polynomial array f stored in global memory is not in a coalesced form (line 7); each thread write to this array with a stride of eight. Besides that, the number of parallel threads (32) is also insufficient to fully harness the computational ability in GPU. Hence, we proposed an optimized parallel CBD implementation, which is presented in Algorithm 15.

Algorithm 14: Pseudocode: Proposed parallel CBD

Input: Byte array $B = b_0, b_1, ..., b_{64\eta-1} \in B^{64\eta}$
Output: Polynomial $f \in R_q$
`/* 32 threads (n/8) are launched in parallel, where tid refers to the ID of each thread. */`
1 $t =$ BytesTo32Bits($B_{4\times tid}$)`// Each thread load four bytes`
2 $d = t \wedge$ `0x55555555`;
3 $d+ = (t \gg 1) \wedge$ `0x55555555`;
4 **for** *j=0; j*< 8*; j++* **do**
5 $\quad a = (d \gg 4 \times j) \wedge$ `0x3`;
6 $\quad b = (d \gg 4 \times j + 2) \wedge$ `0x3`;
7 $\quad f_{8\times tid+j} := a - b$
8 **return** $f_0 + f_1X + ... + f_{255}X^{255}$

Algorithm 15: Pseudocode: Proposed optimized parallel CBD

Input: Byte array $B = b_0, b_1, ..., b_{64\eta-1} \in B^{64\eta}$
Output: Polynomial $f \in R_q$
`/* 256 threads (n) are launched in parallel, where tid refers to the ID of each thread. */`
1 $t =$ BytesTo32Bits($B_{4\times \lfloor tid/8 \rfloor}$)`// Eight threads load four bytes`
2 $d = t \wedge$ `0x55555555`;
3 $d+ = (t \gg 1) \wedge$ `0x55555555`;
4 $a = (d \gg 4 \times (tid\%8)) \wedge$ `0x3`;
5 $b = (d \gg 4 \times (tid\%8) + 2) \wedge$ `0x3`;
6 $f_{tid} := a - b$
7 **return** $f_0 + f_1X + ... + f_{255}X^{255}$

The proposed optimized parallel CBD utilizes 256 threads, wherein every eight threads process four bytes of data cooperatively (line 1); this strategy effectively increases the parallelism. The j loop in Algorithm 14 is now replaced by eight independent threads where their indices are generated through $tid\%8$ (line 4 and 5). By removing the j loop, access to polynomial array f is now fully coalesced, wherein all threads are written to contiguous locations in global memory.

4.5 Symmetric Key Primitives

AES-256 CTR mode was used as the symmetric key primitive to generate random values for XOF and PRF functions. We employed fine-grain AES implementation technique described by Lee et al. [29], wherein four threads compute one AES in parallel. Referring to line 6 in Algorithm 5, XOF is implemented with AES-256 CTR mode, where ρ is used as the encryption key and indices i and j are used as the initialization vectors of counter values. However, this method requires the execution of two **For** loops (line 4 and 5), indicating that multiple launches of the AES kernel are required, which is not optimal for GPU performance.

In this paper, we proposed launching the AES kernel in the GPU only one time to reduce overhead from kernel launches. To achieve this parallel encryption, encryption key ρ needs to be pre-expanded in the CPU before passing to the GPU; we can also utilize thread ID as counter values. In the proposed KEDaaS framework, the public key (and thus ρ) is also known and unchanged. Hence, the encryption key can be pre-expanded only once and stored for future use. A similar strategy can be applied to the PRF function (lines 8, 9 and 11), where r is used as the encryption key and the thread ID is used as a counter value.

Algorithm 16: Fine-grain AES-256 in CTR mode

Input: Pre-exanded encryption key: rk
Input: Four T-box: T_0, T_1, T_2 and T_3 stored in shared memory
Output: Ciphertext array C

/* M threads are launched in parallel, where tid refers to the ID of each thread and $M >= 4$ */
1 __shared__ g_j, srk /* Initialize 32-bit arrays in shared memory */
2 $j \leftarrow tid\%4$/* Compute the fine-grain index */
3 **if** $tid < 60$ **then**
4 $\quad srk_{tid} = rk_{tid}$/* Load expanded keys */
/* Four threads computes one AES, where each thread gets a 32-bit counter */
5 $g_j = \lfloor tid/4 \rfloor$;
6 **for** $i \leftarrow 0$ *to* 13 **do**
7 $\quad g_j = T_0[g_j \gg 24] \oplus T_1[g_{j+1} \gg 16] \oplus T_2[g_{j+2} \gg 8] \oplus T_3[g_{j+3}] \oplus srk_{i*4+j}$
8 $C_{tid} \leftarrow g_j$ /* Each thread stores 32-bit ciphertext */
9 **return** C

Our fine-grain implementation of AES-256 CTR mode is described in Algorithm 16. Four T-boxes are pre-computed and stored in shared memory. Since GPU is a 32-bit architecture, it is optimized to store data in 32-bit format. Hence, the AES internal states and the round keys are initialized as 32-bit array in shared memory. The pre-computed expanded keys are loaded into shared memory as well (line 4). Four threads cooperatively compute one AES-256, where each thread takes a 32-bit counter value as plaintext. The AES round function was performed for 14 rounds (lines 6 -7), in which the T-box and round keys (srk) are involved for encryption. The final results are stored in global memory (line 8). Note that the number of threads involved in AES-256 CTR mode is determined by the amount of random bytes required, which vary for XOF and PRF across different security levels. There are other implementation techniques like coarse grain and bitslice [30], which can compute AES in higher throughput. However, we do not consider that in our implementation because high throughput techniques require a very huge amount of workload to be efficient. Since there are not many random values required by Kyber due to its small polynomial size ($n = 256$), it may not be suitable to use high throughput techniques to compute AES. On the other hand, a fine grain implementation can achieve relatively high throughput and at the same time ensure low latency. However, other lattice-based cryptographic scheme like FrodoKEM may benefit from the high throughput AES implementation.

The KDF and the hash functions (H and G) are not implemented in the GPU because they are serial algorithms that are difficult to parallelized. For this reason, these symmetric primitives are computed entirely in the CPU.

5 Experimental Results and Discussions

This section presents the experimental results and analysis for the proposed implementation techniques to speed up Kyber computation in a GPU. All experiments were performed on a workstation comprised of an Intel i9-9700F CPU (3.3GHz), 16GB RAM and an NVIDIA RTX2060 GPU. The software was written in C with the CUDA SDK v10.2 and operating on a Linux OS (Ubuntu 18.04). An experiment was also performed on Jetson Nano [31], which is an embedded platform that consists of a quad-core ARM A57 CPU with 1.43 GHz and an embedded GPU with 128 cores.

5.1 Performance of Kyber with GPU Acceleration

TABLE 6: Performance of the proposed Kyber implementation on RTX 2060

Batch Size (BS Block)	Encapsulation Throughput (encap/s)			Decapsulation Throughput (decap/s)		
	128-bit	196-bit	256-bit	128-bit	196-bit	256-bit
2	11564	13189	10995	8930	9675	8161
4	29707	24789	22622	20760	16931	16285
8	50602	41630	35544	35626	29498	25371
16	73964	59659	48228	59877	41459	33810
32	103627	78309	65522	69740	54612	45339
64	122309	94331	75245	87527	63881	51117
128	138236	100160	81050	95411	68814	55442
256	148943	105319	86341	100402	72260	58116
512	149813	109769	88857	100837	75821	59755
1024	153327	111222	90375	102955	76173	60798
2048	159388	113869	91199	105932	77736	61267
4096	159668	115300	92090	106225	78487	61728
8192	159898	116158	92319	106360	78889	61854
16384	**160875**	**116428**	**92524**	**106803**	**79039**	**61958**

5.2 Discussions and Comparison with State-of-the-art Work

The GPU acceleration techniques proposed in this paper only apply to Kyber.PKE.Enc (line 4, Algorithm 2) and Kyber.PKE.Dec (line 4, Algorithm 3). Note that Kyber.PKE.Enc is used in both key encapsulation and decapsulation (line 6, Algorithm 3). Other operations including hash functions and the KDF were implemented on the CPU. The GPU implementation employs fine-grain parallelism, in which BS blocks are launched in parallel, each block computing one KEM. In other words, we compute the KEM on the GPU in a batch, where the batch size is the number of blocks (BS), ranging from 2 to 16384.

Referring to Table 6, when the batch size increases, key encapsulation and decapsulation throughput also increase. However, the throughput saturates after the batch size is

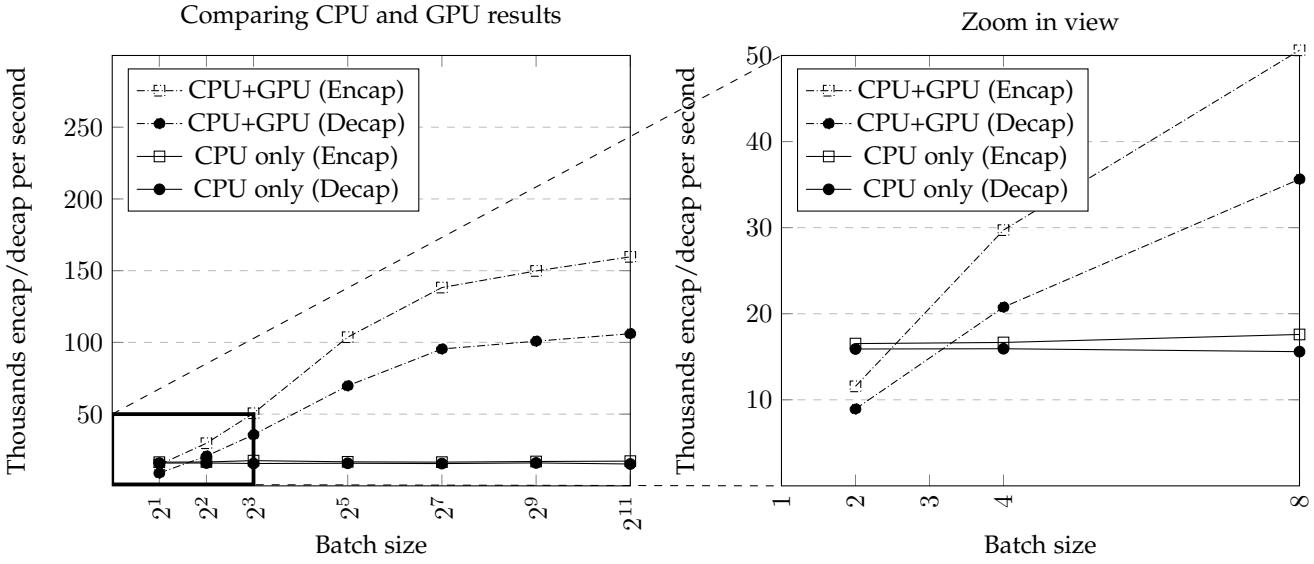

Fig. 5: Comparing the implementation of Kyber512: CPU + GPU vs CPU only (128-bit)

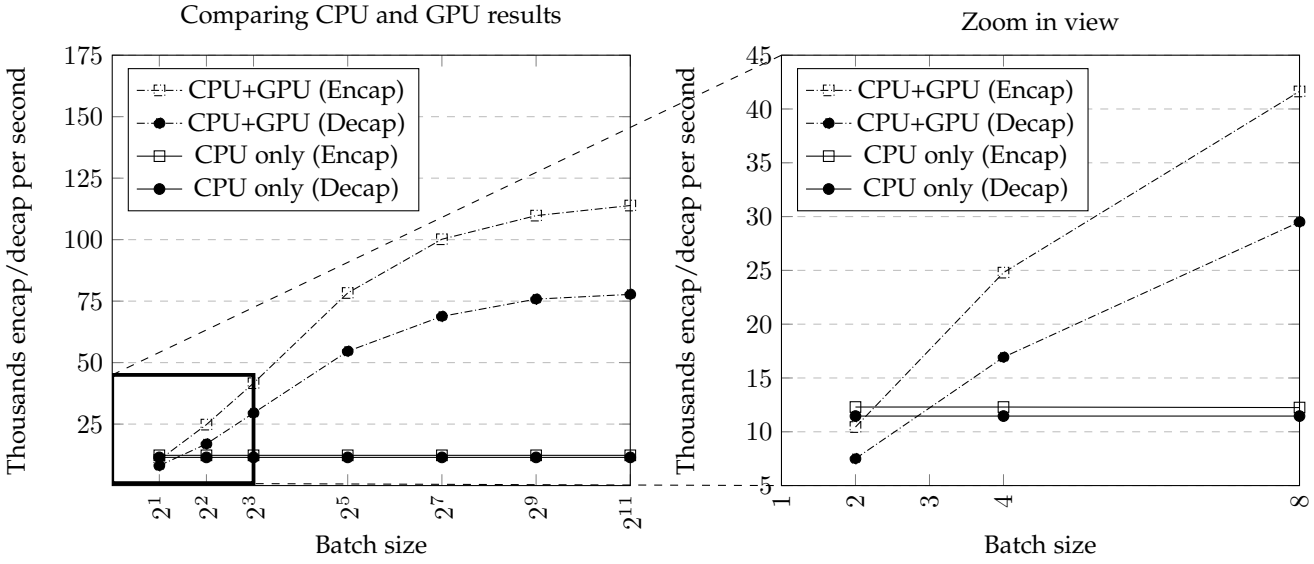

Fig. 6: Comparing the implementation of Kyber768: CPU + GPU vs CPU only (192-bit)

more than 2048, indicating that there is already sufficient workload to keep the GPU busy. Increasing the batch size further does not bring any improvement in throughput, so we do not report the results for $BS > 16384$. The maximum throughput achieved by Kyber with 128-bit security level is more than 100k per second for both encapsulation and decapsulation. This shows that the proposed implementation techniques with acceleration can effectively handle many key encapsulations and decapsulations in a short time, which is useful for IoT applications.

TABLE 7: Comparison with Gupta et al. [22]

Batch Sizeh (BS Block)	Key Exchange per sec.	
64	51.2	6.0
128	56.4	10.0
256	50.2	20.0
1024	61.6	40.0
2048	63.6	90.0
4096	63.8	150.0
8192	63.9	240.0
16384	64.2	370.0

Table 7 shows the comparison with the work by Gupta et al. [22], which is a state-of-the-art implementation of Kyber on GPU. They have implemented Kyber in a coarse grain parallel approach, wherein each thread performs one key exchange, which is essentially a serial implementation. In contrast, we adopted a fine grain approach, in which each block performs one key exchange, so that multiple threads can work cooperatively to speed up the computation. The advantage of our approach is clear when $BS \leq 1024$, as we can produce more key exchanges compared to Gupta et al. However, when the batch size increases, their approach is able to achieve a higher throughput. This comes in the expense of a longer latency as we may need to accumulate

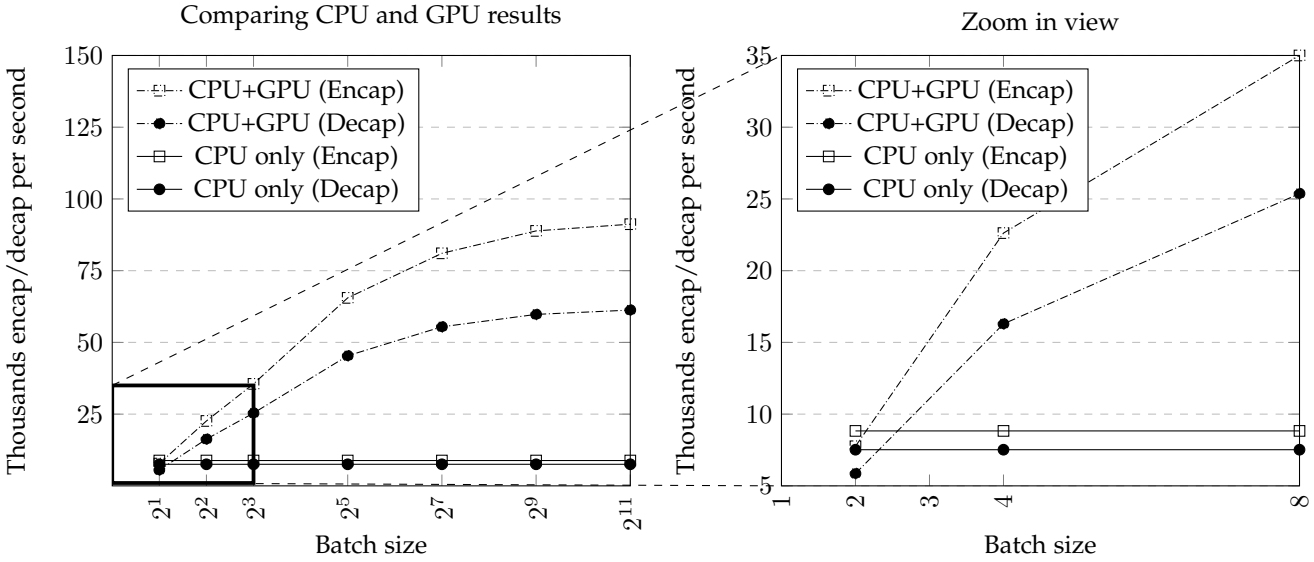

Fig. 7: Comparing the implementation of Kyber1024: CPU + GPU vs CPU only (256-bit)

sufficiently large amount of requests (BS) when using the solution proposed by Gupta et al. In contrast, our proposed implementation can achieve a relatively high key exchange throughput even with a small number of BS. This can be more beneficial for IoT applications where the requests of key encapsulation/decapsulation service is often intermittent and unpredictable. Besides that, our proposed solution can also be combined with the one proposed by Gupta et al. For instance, an acceleration selection unit can be developed to analyze the key exchange request pattern. If the request pattern tends to be in small amount and intermittent, fine grain approach should be utilized to avoid long latency; otherwise, coarse grain approach proposed by Gupta et al. that aims for high throughput should be used.

5.3 Performance Comparison: With and Without GPU Acceleration

To understand the effectiveness of GPU acceleration, we present the throughput of CPU implementation (CPU only) and compare it against the proposed implementation with GPU acceleration (CPU + GPU) on Kyber.PKE.Enc and Kyber.PKE.Dec. Referring to Figure 5, when the batch size is small ($BS = 2$), the CPU implementation showed better throughput. This corresponds to the case where only two blocks are launched in the GPU, and each block performs one KEM with multiple threads; the GPU is heavily underutilized. However, when the batch size increases, more parallel blocks are launched, so we can see that the throughput of CPU+GPU implementation increased significantly. On the other hand, the throughput of the CPU-only implementation remained constant when the batch size increased. Similar performance can be observed for 192-bit and 256-bit implementations, which are presented in Figure 6 and 7.

These experimental results show that the GPU acceleration techniques proposed in this paper can help to elevate KEM performance to a much higher level to enable timely communication in IoT applications. For instance, the highest key encapsulation throughput achieved are 160875 (128-bit), 116428 (192-bit) and 92524 (256-bit), which are respectively 9.11×, 9.47× and 10.48× faster than the CPU alone. Similarly, the highest key decapsulation throughput for our implementations are 106803 (128-bit), 79039 (192-bit) and 61958 (256-bit), which are respectively 6.71×, 6.90×, 8.24× faster than the CPU alone.

Table 8 shows the performance breakdown of Kyber key encapsulation with the proposed GPU acceleration. The results were computed by recording the time for 16384 key encapsulations, and then averaging them to obtain the time consumed by one key encapsulation. The steps to generate matrix $\hat{A}^T$ (lines 4-6 in Algorithm 5) consumes around 6% of the total time. This shows that optimizing the implementation of AES-256 in a GPU is another potential direction to improve the throughput of Kyber. Memory transfer between the CPU and GPU consumed around 2-3% of the total time, while the rest of the operations were almost negligible. Referring to the performance breakdown of Kyber512 (128-bit) before applying GPU acceleration (see Table 3), the majority of the time (89.9%) was spent in Kyber.PKE.Enc. After applying the GPU acceleration techniques, we successfully offloaded most of the computations to the GPU, reducing this to 12.9% of the total time. Since the hash functions and KDF are implemented in the CPU, it is now the main bottleneck, consuming around 87% of the total time for one key encapsulation. However, these operations are hard to parallelize due to the data dependency nature of hash functions; this remains an open issues to be resolved in future work.

5.4 Performance of Kyber Implementation on Embedded GPU

Table 9 shows the results of Kyber on Jetson Nano embedded platform. With $BS = 512$, the encapsulation and decapsulation throughput is $84\times$ to $113\times$ and $75\times$ to $109\times$ lesser than RTX 2060. This is because the GPU in Jetson Nano only has 128 cores compared to 1920 cores in RTX 2060. Moreover, the CPU in Jetson Nano is also executing

TABLE 8: Performance breakdown of Kyber key encapsulation (BS=16384)

Operation	Kyber512 (128-bit)		Kyber768 (192-bit)		Kyber1024 (256-bit)	
	Time (μs)	%	Time (μs)	%	Time (μs)	%
Kyber.PKE.Enc						
gen_at						
AES-256	0.317	5.2	0.414	4.9	0.538	5.0
Rej. Samp.	0.031	0.5	0.046	0.5	0.063	0.6
poly_getnoise						
AES-256	0.059	1.0	0.087	1	0.101	0.9
CBD	0.037	0.6	0.053	0.6	0.054	0.5
NTT	0.017	0.3	0.023	0.3	0.027	0.3
INVNTT	0.030	0.5	0.034	0.4	0.042	0.4
Miscellaneous	0.109	1.8	0.16	1.9	0.205	1.9
Mem. Trans.	0.192	3.1	0.241	2.9	0.284	2.6
Hash + KDF (CPU)	5.352	87.1	7.371	87.4	9.464	87.8
Total	6.144	100.0	8.429	100.0	10.778	100.0

TABLE 9: Performance of the proposed Kyber implementation on Jetson Nano [31]

Batch Size (BS Block)	Encapsulation Throughput (encap/s)			Decapsulation Throughput (decap/s)		
	128-bit	196-bit	256-bit	128-bit	196-bit	256-bit
2	418	451	398	205	198	185
4	602	676	512	335	384	309
8	718	788	753	509	498	442
16	835	892	796	688	625	438
32	968	996	885	799	681	539
64	1095	999	891	805	786	702
128	1111	1003	893	855	808	751
256	1241	1115	1014	910	888	792
512	**1345**	**1109**	**1054**	**918**	**895**	**795**

in a lower frequency (1.43GHz vs 3.3GHz). Although the throughput of Kyber KEM on embedded platform is not very high, it is sufficient to be used as a gateway device to encapsulation and decapsulation multiple keys in real time to serve the local sensor nodes.

6 Conclusion

Security issues in IoT applications have become more prominent recently, due to the rapid adoption of the IoT in various field. One of the key methods to secure IoT communications is to frequently refresh the symmetric encryption keys, which are used in encrypting sensor data. This paper first proposes an offloading architecture, wherein the symmetric keys are encapsulated/decapsulated by a third party that provide KEDaaS, which is an emerging buiness model. On top of this, various optimization techniques were proposed to improve the implementation of Kyber, a KEM currently under review by Round 3 NIST standardization process [9]. The proposed techniques optimized the performance of NTT, rejection sampling and CBD operations implemented in a GPU, achieving 162760 encapsulations per second and 107631 decapsulations per second from an RTX2060 GPU. This high throughput implementation allows KEDaaS to be used in IoT systems, providing timely key encapsulation/decapsulation without putting much of a burden on the IoT sensor nodes.

Side-channel analysis on GPU has become an important research topic recently. For instance, Naghibijouybari et al. [33] had exploited the vulnerabilities in CUDA driver to fingerprint the website and neural network model accurately. Following this work, NVIDIA had restricted the use of hardware profiling counters as an effort to mitigate this kind of attacks. Another interesting work was presented by Gao et al. [32], in which electromagnetic (EM) leakage models targeting GPU were analyzed and experimented on a table-based AES implementation. Our AES implementation is also using look-up tables, which may be vulnerable to this kind of attacks. However, this side-channel attack [32] requires close proximity to the particular GPU that resides in a server farm in order to collect the EM signals. This may not be a practical threat as it is not easy to locate which GPU is computing the Kyber KEM, due to the virtualization technology widely used in the cloud computing services. Nevertheless, such potential vulnerability should be addressed on embedded GPU (Jetson Nano), which is a promising direction for future work. On the other hand, all other operations in Kyber like NTT and random sampling are implemented in constant time, so we believe that they are not vulnerable to timing attacks. A more thorough analysis to develop Kyber implementation that resists side-channel attacks is another interesting research direction that we plan to pursue.

Acknowledgment

Wai-Kong Lee and Seong Oun Hwang were supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2019H1D3A1A01102607, 2020R1A2B5B01002145).

References

[1] Statistica. (2018). Internet of Things (IoT) Connected Devices InstalledBase Worldwide From 2015 to 2025 (in Billions). [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, accessed at 15 July 2020.

[2] F. Cirillo, D. Gómez, L. Diez, I. E. Maestro, T. B. J. Gilbert, R. A., "Smart City IoT Services Creation Through Large-Scale Collaboration", IEEE Internet of Things Journal, vol. 7, no. 6, pp. 5267-5276, 2020.

[3] M. Ayaz, M. A.-Uddin, Z. Sharif, A. Mansour, E.-H. M. Aggoune, "Internet-of-Things (IoT)-Based Smart Agriculture: Toward Making the Fields Talk", IEEE Access, vol. 7, pp. 129551-129583, 2019.

[4] F. Tao and Q. Qi, "New IT Driven Service-Oriented Smart Manufacturing: Framework and Characteristics", IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 49, no. 1, pp. 81-91, 2019.

[5] S. Patranabis, N. Datta, D. Jap, J. Breier, S. Bhasin, D. Mukhopadhyay, "SCADFA: Combined SCA+DFA Attacks on Block Ciphers with Practical Validations", IEEE Transactions on Computers, vol. 68, no. 10, pp. 1498-1510, 2019.

[6] A. Singh, N. Chawla, J. H. Ko, M. Kar, S. Mukhopadhyay, "Energy Efficient and Side-Channel Secure Cryptographic Hardware for IoT-Edge Nodes", IEEE Internet of Things Journal, vol. 6, no. 1, pp. 421-434, 2019.

[7] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu, "Security vulnerabilitiesof Internet of Things: A case study of the smart plug system", IEEE Internet of Things Journal, vol. 4, no. 6, pp. 1899–1909, 2017.

[8] C. Cheng, R. Lu, A. Petzoldt, and T. Takagi, "Securing the Internetof Things in a quantum world", IEEE Communication Magazine, vol. 55, no. 2, pp. 116–120, Feb. 2017.

[9] Post-Quantum Cryptography: Round 3 Submissions. (2020) [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions, accessed at 25 July 2020.

[10] W.-K. Lee, S. Akleylek, D. C.-K. Wong, W.-S. Yap, B.-M. Goi, S.-O. Hwang, "Parallel Implementation of Nussbaumer Algorithm and Number Theoretic Transform on a GPU Platform: Application to qTESLA", Journal of Supercomputing, in press, 2020.

[11] V. Varadharajan and U. Tupakula, "Security as a service modelfor cloud environment", IEEE Transactions on Network Service Management, vol. 11,no. 1, pp. 60–75, Mar. 2014.

[12] S. Sun, R. Zhang, H. Ma, "Efficient Parallelism of Post-Quantum Signature Scheme SPHINCS", IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 11, 2020.

[13] C. Tian, J. Yu, H. Zhang, H. Xue, C. Wang, K. Ren, "Novel Secure Outsourcing of Modular Inversion for Arbitrary and Variable Modulus", IEEE Transactions on Services Computing, in press, 2019.

[14] C. C. Chang, W. K. Lee, Y. Liu, B. M. Goi, Raphael C.-W. Phan "Signature gateway: Offloading signature generation to IoT gateway accelerated by GPU", IEEE Internet of Things Journal, vol. 6, no. 3, pp. 4448-4461, 2018.

[15] W. Dai, J. Schanck, B. Sunar, W. Whyte, Z. Zhang, "NTRU modular lattice signature scheme on CUDA GPUs", 11th International Workshop on Security and High Performance Computing Systems (SHPCS), 2016.

[16] J. Hoffstein, J. Pipher, J. Schanck, J. Silverman, W. Whyte, "Transcript secure signatures based on modularlattices," Post-Quantum Cryptography (PQCrypto 2014), Lecture Notes in Computer Science, vol. 8772, pp.142–159, 2014.

[17] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, Z. W.-O'Hearn, "SPHINCS: Practical stateless hash-based signatures," Proceedings of Annual International Conference of Theory and Applications of Cryptographic Techniques (EUROCRYPT2015), pp. 368–397, 2015.

[18] N. Emmart, and C. C. Weems, "High precision integer multiplication witha GPU using Strassen's algorithm with multiple FFT sizes," Parallel Processing Letters, vol. 21, no. 3, pp. 359–375, 2011.

[19] W. Wang, Y. Hu, L. Chen, X. Huang, B. Sunar, "Exploring the feasibility of fully homomorphic encryption," IEEE Transactions on Computers, vol. 64, no. 3, pp. 698-706, 2013.

[20] S. Akleylek, Z. Y. Tok, "Efficient arithmetic for lattice-based cryptography on GPU using the CUDA platform," 22nd IEEE Signal Processing and Communications Applications Conference (SIU), Trabzon, 2014.

[21] S. Akleylek, O. Dagdelen, Z. Y. Tok, "On the Efficiency of Polynomial Multiplication for Lattice-Based Cryptography on GPUs Using CUDA," International Conference on Cryptography and Information Security in the Balkans, pp 155-168, Koper, 2016.

[22] N. Gupta, A. Jati, A. K. Chauhan and A. Chattopadhyay, "PQC Acceleration Using GPUs: FrodoKEM, NewHope, and Kyber", IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, 2021.

[23] L. Botros, M. J. Kannwischer, P. Schwabe, "Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4", International Conference on Cryptology in Africa (AFRICACRYPT 2019), Lecture Notes in Computer Science, vol. 11627. Springer, Cham, 2019.

[24] Z. Chen, Y. Ma, T. Chen, J. Lin, J. Jing, "Towards Efficient Kyber on FPGAs: A Processor for Vector of Polynomials", 25th Asia and South Pacific Design Automation Conference (ASP-DAC), Beijing 2020.

[25] U. Banerjee, T. S. Ukyab, A. P. Chandrakasan, "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols," IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019, vol. 4, pp. 17-61, 2020.

[26] C. Liu, Y. Cao, Y. Luo, G. Chen, V. Vokkarane, M. Yunsheng, S. Chen and P. Hou, "A New Deep Learning-Based Food Recognition System for Dietary Assessment on An Edge Computing Service Infrastructure", IEEE Transactions on Services Computing, vol. 11, no. 2, 2018.

[27] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," Designs, Codes and Cryptography, vol. 75, no. 3, pp. 565–599, 2015.

[28] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, D. Stehlé, "CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation (version 2.0)". [Online] Available: https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf, accessed at 25 July 2020.

[29] W.K. Lee, B.M. Goi, Raphael R.-C. Phan, "Terabit encryption in a second: Performance evaluation of block cipher in GPU with Kepler, Maxwell, and Pascal architectures", Concurrency and Computation: Practice and Experience, vol. 31, pp. e5048, 2019.

[30] O. Hajihassani, S. K. Monfared, S. H. Khasteh, S. Gorgin, "Fast AES Implementation: A High-Throughput Bitsliced Approach", IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 10, 2019.

[31] Jetson Nano Developer Kit. [Online] Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit, accessed at 5 August 2020.

[32] Y. Gao and Y. Zhou, "Side-Channel Attacks With Multi-Thread Mixed Leakage", IEEE Transactions on Information Forensics and Security, vol. 16, 2021.

[33] H. Naghibijouybari, A. Neupane, Z. Qian and N. Abu-Ghazaleh, "Rendered insecure: GPU side channel attacks are practical," Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, 2018.

Wai-Kong Lee received the B.Eng. degree in electronics and the M.Eng.Sc. degree from Multimedia University in 2006 and 2009, respectively. He received the Ph.D. degree in engineering from Universiti Tunku Abdul Rahman, Malaysia in 2018. He was a Visiting Scholar with Carleton University, Canada, in 2017, Feng Chia University, Taiwan, in 2016 and 2018, and OTH Regensburg, Germany, in 2015, 2018 and 2019. Prior to joining academia, he worked in several multi-national companies including Agilent Technologies (Malaysia) as R&D engineer. His research interests are in the areas of cryptography, numerical algorithms, GPU computing, Internet of Things, and energy harvesting. He is currently a post-doctoral researcher in Gachon University, South Korea.

Seong Oun Hwang received the B.S. degree in mathematics from Seoul National University, in 1993, the M.S.degree in information and communications engineering from the Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, South Korea. He worked as a Software Engineer with LG-CNS Systems,Inc., from 1994 to 1996. He also worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007. He worked as a Professor with the Department of Software and Communications Engineering, Hongik University, from 2008 to 2019. He is currently a Professor with the Department of Computer Engineering, Gachon University. He is also an Editor of ETRI Journal. His research interests include cryptography, cybersecurity, and artificial intelligence