DPCrypto: Acceleration of Post-Quantum Cryptography Using Dot-Product Instructions on GPUs

Wai-Kong Lee, *Member, IEEE*, Hwajeong Seo, *Member, IEEE*, Seong Oun Hwang, *Senior Member, IEEE*, Ramachandra Achar, *Fellow, IEEE*, Angshuman Karmakar, and Jose Maria Bermudo Mera

***Abstract*— Modern NVIDIA GPU architectures offer dot-product instructions (DP2A and DP4A), with the aim of accelerating machine learning and scientific computing applications. These dot-product instructions allow the computation of multiply-and-add instructions in a single clock cycle, effectively achieving higher throughput compared to conventional 32-bit integer units. In this paper, we show that the dot-product instruction can also be used to accelerate matrix-multiplication and polynomial convolution operations, which are widely used in post-quantum lattice-based cryptographic schemes. In particular, we propose a highly optimized implementation of FrodoKEM wherein the matrix-multiplication is accelerated by the dot-product instruction. We also present specially designed data structures that allow an efficient implementation of Saber key-encapsulation mechanism, utilizing the dot-product instruction to speed-up the polynomial convolution. The proposed FrodoKEM implementation achieves 4.37× higher throughput than the state-of-the-art implementation on a V100 GPU. This paper also presents the first implementation of Saber on GPU platforms, achieving 124,418, 120,463, and 31,658 key exchanges per second on RTX3080, V100, and T4 GPUs, respectively. Since matrix-multiplication and polynomial convolution operations are the most time-consuming operations in lattice-based cryptographic schemes, we strongly believe that the proposed methods can be beneficial to other KEM and signatures schemes based on lattices.**

***Index Terms*— Post-quantum cryptography, dot-product, polynomial convolution, matrix-multiplication, graphics processing unit, FrodoKEM and Saber.**

Manuscript received February 15, 2022; revised April 21, 2022; accepted May 15, 2022. The work of Wai-Kong Lee was supported by the Brain Pool Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information Communication Technology (ICT) under Grant 2019H1D3A1A01102607. The work of Seong Oun Hwang was supported by the NRF funded by the Ministry of Science and ICT under Grant 2020R1A2B5B01002145. The work of Ramachandra Achar was supported in part by the Natural Science and Engineering Research Council of Canada (NSERC). The work of Angshuman Karmakar was supported by Research Foundation– Flanders (FWO) as a Junior Post-Doctoral Fellow under Grant 203056/1241722N LV. This article was recommended by Associate Editor R. Azarderakhsh. *(Corresponding author: Seong Oun Hwang.)*

Wai-Kong Lee and Seong Oun Hwang are with the Department of Computer Engineering, Gachon University, Seongnam 13120, South Korea (e-mail: bardic@naver.com).

Hwajeong Seo is with the Department of Computer Engineering, Hansung University, Seoul 02876, South Korea.

Ramachandra Achar is with the Department of Electronics, Carleton University, Ottawa, ON K1S 5B6, Canada.

Angshuman Karmakar and Jose Maria Bermudo Mera are with the COSIC, KU Leuven, 3000 Leuven, Belgium.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2022.3176966.

Digital Object Identifier 10.1109/TCSI.2022.3176966

I. Introduction

In 2016, the National Institute of Standards and Technology (NIST) initiated a standardization process to select public-key encryption algorithms [1], both key-encapsulation mechanism (KEM) and digital signature schemes, that are resistant to quantum computing attacks. This is an appropriately timed response to the threat of quantum computers that can break existing public key cryptographic algorithms. This standardization process has stimulated a lot of interest in the post-quantum cryptography (PQC) with a focus on improving the security of PQC algorithms and the performance of their implementations. Currently, the standardization process is in the third round, where 15 candidates have been selected [1] out of 69 submissions from the first round. Among these 15 third round candidates, seven of them are based on lattice hard problems. One of the main performance bottleneck in lattice-based cryptography is the polynomial convolution or matrix multiplication. Some schemes, e.g., Kyber [2] and Dilithium [3], are based on special ring structures that allow the polynomial convolution to be computed efficiently using Number Theoretic Transform (NTT). However, other schemes that do not have such a ring structure, e.g., FrodoKEM [4] and Saber [5], require carefully designed implementations in order to achieve reasonably fast performance.

GPUs are a massivley parallel computing architecture which are used for dedicated graphics processing. However, such parallel architecture can also be exploited for speeding up parallel non-graphics computation. Due to this reason, GPUs are widely used to speed-up algorithms in various domains including deep learning [6] and healthcare [7]. Recently, there have been several attempts on utilizing GPUs for implementing cryptographic algorithms. For instance, attempts to accelerate homomorphic encryption using GPUs were presented by Al Badawi *et al.* [8]. Besides that, GPUs have also been used to implement symmetric-key cryptographic algorithms [9], [10], achieving high throughput.

Since the commencement of NIST standardization process, there have been some research works that explore the possibility of accelerating PQC with GPUs. One of the most notable works was presented by Sun *et al.* [11], where the authors exploit the parallel architecture of a GPU to implement the tree structure of the SPHINCS signature scheme. However, SPHINCS was not selected to the third round of the NIST

standardization process. Gupta *et al.* [12] presented a comprehensive benchmark of FrodoKEM, NewHope, and Kyber KEMs on various GPU platforms. Recently, Lee *et al.* [13] and Gao *et al.* [14] also provided high throughput implementations of Kyber and NewHope KEM on GPUs. These prior works are able to achieve a high throughput implementation by using GPU as an accelerator, but they only focus on algorithmic parallelization and low level optimization, without using the advanced features found in modern GPU architectures.

A. Motivations

In a recent work, Lee *et al.* [15] introduced some techniques to compute polynomial convolution and matrix-multiplication using tensor core in GPUs. The key idea is to pack many polynomials into a matrix form and compute them efficiently using the tensor core in a GPU. Although performance improvements are impressive, this technique [15] can only achieve its full benefit if the usage of non-ephemeral key is permitted. Moreover, it relies on fast tensor core that supports half precision, which implies that it cannot be used by lattice-based cryptography schemes that have a large modulus ($q > 2,048$). Note that most of the finalists in NIST PQC [1] have large modulus $q > 2,048$. In particular, FrodoKEM ($q = 32,768$ or $q = 65,536$) and Saber ($q = 8,192$ and $p = 1,024$) cannot benefit from the tensor-core-based solution. In this paper, we fill this research gap by proposing novel implementation techniques using dot-product instructions on GPUs, which can be applicable to larger modulus sizes.

Dot-product is a widely used operation found in many different algorithms. Due to this reason, many research works have been devoted to design specific hardware for dot-product computation [16], [17], aimed towards providing a faster and more energy efficient implementations. The popular processor architecture like ARM has released special instructions [18] to handle dot-product operation. Unsurprisingly, NVIDIA also introduced DP4A and DP2A dot-product instructions into its Pascal architecture GPU [19]. The DP2A instruction is particularly useful in computing polynomial convolution and matrix-multiplication found in FrodoKEM and Saber KEM, which are NIST PQC Round-3 candidates. However, utilizing dot-product instructions to compute these operations in parallel is not straightforward. Naïve implementations could lead to serious overhead in loading/storing intermediate results.

B. Contributions

This paper is the first implementation that utilizes the dot-product instruction in GPU architectures to accelerate lattice-based cryptography. Proposed techniques can achieve a higher performance on various modern GPU architectures compared to other state-of-the-art works that only rely on conventional 32-bit integer units. We summarize our contributions below:

1) We propose a highly optimized technique for matrix-multiplication implementation on GPU. We use the DP2A instruction to speed-up dot-product operations between two matrices. The proposed technique is able to accelerate the matrix-multiplication up-to 1.37×, 1.83×, and 1.58× compared to the existing implementations with conventional 32-bit integer units, i.e., without dot-product instructions, on RTX3080, V100, and T4 GPUs, respectively. This dot-product aided technique has been applied to FrodoKEM, i.e., DPFrodo, achieving a 4.37× speed-up compared to the state-of-the-art implementation of FrodoKEM [12] on a V100 GPU platform.
2) In this paper, we present the first optimized implementation of Saber on GPUs using 32-bit integer units. To further improve the performance, we also incorporate the dot-product technique. We found that the polynomial convolution in Saber requires the coefficients to be read in a cyclic form. A naive implementation of parallel polynomial convolution using dot-product instruction introduces excessive conditional statements, which is far from efficient. To overcome these problems, we proposed a novel data structure that reduces conditional statements and allow fully coalesced global memory access. When applied to the polynomial convolution, i.e., matrix-vector multiplication, in Saber, the proposed technique with dot-product instructions can achieve up-to 1.63×, 1.28×, and 1.54× higher throughput compared to the proposed implementation using 32-bit integer units, i.e., without dot-product instructions, on RTX3080, V100, and T4 GPUs, respectively. The dot-product aided technique was applied to Saber (DPSaber) parameter set with $N = 256$ and $l = 3$, and the achieved key exchange throughputs are 124,418, 120,463, and 20,225 on RTX3080, V100, and T4 GPUs, respectively.
3) The proposed dot-product aided techniques are suitable to be used in lattice-based cryptographic schemes that cannot leverage the NTT directly to speed-up the polynomial convolution. It is also beneficial for schemes that utilize a larger modulus ($2,048 < q \leq 65,536$), which cannot be achieved by the tensor-core-based solution proposed by Lee *et* . [15]. Proposed DPFrodo and DPSaber implementations support both ephemeral and non-ephemeral key usage, which is more flexible compared to Lee *et al.* [15]. It supports high throughput KEM, which is beneficial to conventional client-server based Internet communication, as well as the emerging Internet of Things (IoT) applications.

In our implementations, all the computations in encapsulation and decapsulation are executed on the GPU device. All the implementations discussed in this paper are available publicly at https://github.com/benlwk/DPCrypto.

II. Background

In this section, we provide a brief introduction to the selected lattice-based cryptographic schemes, i.e., FrodoKEM and Saber, and the related hard problems. For more details, we refer to the very detailed specification documents [4], [5] of these two schemes. Following this, we also present a summary of key features in modern NVIDIA GPU architectures.

A. FrodoKEM

FrodoKEM is a lattice-based KEM that relies on the hardness of learning with errors (LWE) problem [20]. It was

firstly introduced as a key exchange protocol in [21] and later on developed into a KEM for submission to the NIST standardization process. FrodoKEM was selected as an alternate candidate in Round-3 of NIST PQC standardization. The main performance bottleneck in FrodoKEM is the matrix multiplication [22], wherein the matrix elements are $\mathbb{Z}_q$.

B. Saber

Saber [5] is a lattice-based KEM which is based on module-lattices. Unlike most other lattice-based cryptosystems, the security of Saber is based on learning with rounding problem [23] rather than the learning with errors [20]. The advantage of the former over the later is that the error term is generated inherently due to rounding in the former case whereas it needs to be added explicitly in the later case. This results in lesser requirements of pseudo-random numbers which leads to better efficiency than other schemes. Saber is a one of four finalist candidates in the KEM category of the NIST's standardization procedure. Similar to other lattice-based KEM schemes, it has been shown that the polynomial multiplication is the most computationally expensive component [24]–[27] of Saber. The polynomial multiplication in Saber is used to compute the matrix-vector multiplication and inner product. Unlike FrodoKEM, Saber operates over quotient ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$.

C. Overview of NVIDIA GPU Architectures

1) CUDA Programming Model: A GPU hardware has multiple execution units named *Streaming Multiprocessors* (SMs), where each SM hosts hundreds of CUDA cores. For instance, the RTX3080 is an Ampere architecture GPU with 68 SMs, each SM consists of 128 cores. CUDA is the Software Development Kit released by NVIDIA to ease programming of GPU for general purpose computing. Under the CUDA programming model, multiple threads are grouped into a block, where multiple blocks form a GPU grid. This relationship is illustrated in Figure 1, where each thread and block can be indexed individually for parallel computing. NVIDIA GPUs grouped 32 threads into one *warp* in order to allow efficient instruction scheduling and memory access. *Warp divergence* occurs if threads within a warp do not execute the same path, which may have serious performance penalty. In the subsequent presentation, we refer *tid* as unique the ID for parallel threads within a block.

2) Memory Hierarchy: There are two types of GPU memory in general, which has a huge difference in performance: on-chip and off-chip memory. *Global memory* is essentially the DRAM, i.e., off-chip memory, which is large in size but slow in performance. The use of global memory is unavoidable in most of the situations, as one needs to share the data between the CPU and GPU. To achieve a high performance in global memory, the read/write must be performed in contiguous memory locations. This allows the memory access to be performed in burst mode in DRAM. *Shared memory* is only accessible by threads within the same block, but it is a user-managed cache, which has better performance compared to global memory. *Register* is the fastest memory in a GPU; it has a very limited size, e.g., 64K words per SM for the RTX3080.

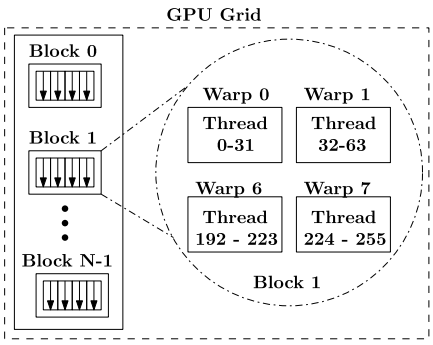

Fig. 1. Relationship between grid, block, warp, and thread in CUDA.

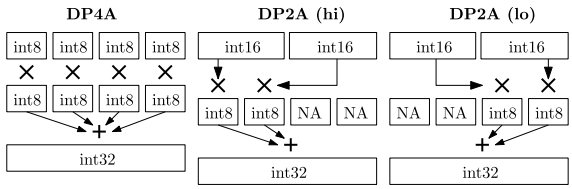

Fig. 2. Dot-product instructions in NVIDIA GPU.

3) Dot-Product Instructions: Dot-product instructions were first introduced into Pascal architecture, i.e., compute capability 6.1, and they have been supported in the subsequent GPU architectures. Referring to Figure 2, there are two versions of dot-product instructions in NVIDIA GPUs. DP4A supports 4-way dot-product operations on four 8-bit inputs, where the result is accumulated on a 32-bit integer. Similarly, DP2A allows 2-way dot-product operations on two 16-bit inputs with another two 8-bit inputs; the result is also accumulated on a 32-bit integer. The two 8-bit inputs comes from either the first or the last two bytes (see DP2A (hi) and DP2A (lo) in Figure 2). Both versions support signed and unsigned operands.

D. Related Work

Efficient implementation of PQC on various platforms is an emerging research area in the past decade. For instance, Seo *et al.* [28] presented an optimized implementation of SIKE on ARM processor. Zhu *et al.* [29] shows that Karatsuba algorithm can be used to design a high-speed multiplier hardware architecture for Saber KEM.

The first FrodoKEM implementation on GPU was presented by Gupta *et al.* [12]. Authors utilized *single* mode to compute FrodoKEM, wherein multiple blocks and multiple threads cooperatively execute algorithms on a GPU. This involves the use of atomic instructions to avoid data hazard introduced by parallel read/write from different blocks. They also proposed a tiling technique to compute the matrix-matrix multiplication, efficiently. However, their implementation does not show high throughput performance, due to high amount of atomic instructions. Moreover, FrodoKEM can be computed by using 16-bit coefficients, but Gupta *et al.* [12] only utilized the 32-bit integer units, which is not an optimal choice.

Another two notable works published recently are from Lee *et al.* [13] and Gao *et al.* [14]. These works showcased high throughput implementations of Kyber and NewHope KEM on GPU platforms, which relies on the use NTT [30].

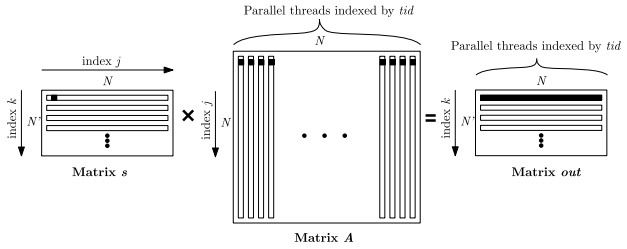

Fig. 3. Parallel implementation of matrix multiplication in FrodoKEM.

However, these works also do not use advanced features, e.g., dot-product instructions, found in modern GPU architectures.

III. GPU Implementation Techniques

In this section, we present details of our GPU implementation techniques targeting two selected lattice-based cryptographic schemes. We believed that our methods are applicable to other lattice-based cryptographic schemes with small appropriate changes.

A. Highly Optimized (INT32) and Dot-Product Aided (DPFrodo) Matrix Multiplication Implementations of FrodoKEM on GPUs

Matrix multiplication is one of the most time consuming operations in FrodoKEM. Referring to Figure 3, the matrix multiplication in key-encapsulation and decapsulation involves a rectangular matrix, i.e., $N \times N'$, and a square matrix, i.e., $N \times N$. This can be implemented on a GPU through the following steps described in Algorithm 1. The matrices *out*, *A* and *s* are initially stored in the global memory. Firstly, N values are loaded in parallel from the matrix A (line 5). Next, the algorithm executes k loop (lines 6 $\sim$ 8), wherein values from public matrix A are multiplied with a value from secret matrix *s*, and then added to results in matrix *out*. Same steps are repeated for N times to complete the j loop in lines 4 $\sim$ 9. This process is also illustrated in Figure 3, in which highlighted portions represent the parallel execution of lines 4 and 6 on a GPU. Note that in line 6, the results of multiplications are accumulated on matrix *out*, which is stored in the global memory. This naïve implementation causes a lot of read/write into the global memory, seriously limiting the performance of implementation on a GPU.

A closer look into FrodoKEM reveals that the parameter N' is small, e.g., $N' = 8$ across all three proposed parameter sets, compared to parameter N [4]. Hence, it is possible to fully unroll the k loop in Algorithm 1. Algorithm 2 shows this improved implementation technique. By doing this, we can compute eight multiply-and-accumulate (MAC) operations in one iteration of j loop, and store all the intermediate results in registers *sum0* to *sum7*. This allows us to exploit the use of fast registers and avoid excessive read/write to the slow global memory when computing the MAC operations. This highly optimized matrix multiplication technique relies on INT32 to perform the matrix multiplications and accumulations. We applied this technique to the GPU implementation of FrodoKEM.

Next, we explore how to improve the INT32 version of matrix multiplication using dot-product instructions. Referring to Table I, the modulus q is either 32,768 or 65,536. This implies that the matrix multiplications and accumulations can be carried out entirely on a 16-bit variable without causing any errors. Note that the matrix A was originally stored in column major order after the random sample generation using AES or SHAKE [4]. To perform matrix-matrix multiplication, matrix A is read in a row major order; this is illustrated in Figure 4. When the proposed DPFrodo is used, two matrix elements are packed into one register, which can be indexed as x or y component. To access these packed elements in Matrix A, even-indexed threads load only x components, while odd-indexed threads load only y components. For the smaller Matrix s (see Algorithm 3), the packing is more straightforward.

Algorithm 1 Parallel Matrix Multiplication in FrodoKEM

```
1: procedure MAT_MUL_AND_ADD(out, A, s)
2:     sum = 0;
3:     for (j = 0; j < N; j++) do  ▷ N × N' is the size of matrix
4:         load_a = A[j × N + tid];
5:         for (k = 0; k < N'; k++) do
6:             out[k × N + tid] += load_a × s[k × N + j];
7:         end for
8:     end for
9: end procedure
```

Algorithm 2 Unrolled Parallel Matrix Multiplication in FrodoKEM

```
1: procedure MAT_MUL_UNROLL(out, A, s)
2:     sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
3:     sum4 = 0, sum5 = 0, sum6 = 0, sum7 = 0;
4:     for (j = 0; j < N; j++) do
5:         load_a = A[j * N + tid];
6:         sum0 += load_a × s[j];          ▷ Unroll 8 times (N')
7:         sum1 += load_a × s[1 × N + j];
8:         sum2 += load_a × s[2 × N + j];
9:         sum3 += load_a × s[3 × N + j];
10:        ...                             ▷ Removed for brevity
11:    end for
12:    out[tid] = sum0;                    ▷ Unroll 8 times (N')
13:    out[1 × N + tid] = sum1;
14:    out[2 × N + tid] = sum2;
15:    out[3 × N + tid] = sum3;
16:    ...                                 ▷ Removed for brevity
17: end procedure
```

Algorithm 3 DPFrodo: Packing the Matrix *s*

```
1: procedure PACK_MAT_S(s_packed, s)
2:     for (i = 0; i < N'; i++) do ▷ N' is 8 for FrodoKEM
3:         s_packed[i × N/2 + tid].x = s[i × N + 2 × tid];
4:         s_packed[i × N/2 + tid].y = s[i × N + 2 × tid + 1];
5:     end for
6: end procedure
```

TABLE I
OVERVIEW OF FRODOKEM [4] AND SABER [5] PARAMETERS

Algorithm	Underlying Hard Problem	Modulus (q)	Dimension of polynomial/matrix	NIST PQC Round-3
FrodoKEM-640	Standard LWE	2^{15}	$N = 640$, $N' = 8$	Alternate candidate
FrodoKEM-976		2^{16}	$N = 976$, $N' = 8$	
FrodoKEM-1344		2^{16}	$N = 1344$, $N' = 8$	
LightSaber	Module LWE	2^{13}	$N = 256$, $l = 2$	Finalist
Saber			$N = 256$, $l = 3$	
FireSaber			$N = 256$, $l = 4$	

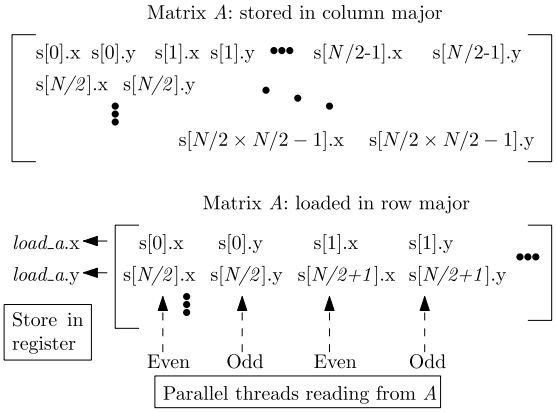

Fig. 4. Parallel implementation of matrix multiplication in FrodoKEM.

The proposed parallel matrix-matrix multiplication in FrodoKEM using the dot-product instruction is detailed in Algorithm 4. Similar to Algorithm 2, the matrices *out*, *A* and *s* are initially stored in the global memory. The *j* (lines 4 $\sim$ 20) loop is executed to accumulate the results of matrix multiplication. In each iteration, even-index threads load *x* components of matrix *A* (lines 5 $\sim$ 7), while odd-index threads load *y* components (lines 8 $\sim$ 10). This follows by multiplications between matrix *A* and *s* (lines 12 $\sim$ 19), which is fully unrolled by $N' = 8\times$. The accumulations are performed on registers *sum0* to *sum7* to exploit their fast read/write speed. Finally, results are stored in the output array (lines 21 $\sim$ 25). Note that the *j* loop in Algorithm 4 is now reduced by half compared to Algorithm 2, due to the use of dot-product instructions (DP2A) that computes two 16-bit MACs in one cycle.

Algorithm 4 DPFrodo: Parallel Matrix Multiplication in FrodoKEM With Dot-Product Instruction

```
1: procedure MAT_MUL_AND_ADD_UNROLL(out, A, s)
2:     sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
3:     sum4 = 0, sum5 = 0, sum6 = 0, sum7 = 0;
4:     for (j = 0; j < N/2; j++) do
5:         if tid%2 == 0 then
6:             load_a.x = A[j * N + tid/2].x;
7:             load_a.y = A[j * N + N/2 + tid/2].x;
8:         else
9:             load_a.x = A[j * N + tid/2].y;
10:            load_a.y = A[j * N + N/2 + tid/2].y;
11:        end if
12:        sum0 +=load_a.x × s[j].x + load_a.y × s[j].y;
13:        sum1 +=load_a.x × s[1 × N + j].x+
14:            load_a.y × s[1 × N + j].y;
15:        sum2 +=load_a.x × s[2 × N + j].x+
16:            load_a.y × s[2 × N + j].y;
17:        sum3 +=load_a.x × s[3 × N + j].x+
18:            load_a.y × s[3 × N + j].y;
19:        ...        ▷ Unroll 8 times (N'). Removed for brevity
20:    end for
21:    out[tid] = sum0;
22:    out[1 × N + tid] = sum1;
23:    out[2 × N + tid] = sum2;
24:    out[3 × N + tid] = sum3;
25:    ...        ▷ Unroll 8 times (N'). Removed for brevity
26: end procedure
```

B. The First Optimized (INT32) and Dot-Product Aided (DPSaber) Polynomial Convolution Implementations of Saber on GPUs

The most time consuming operation in Saber is the polynomial convolution [27], [31], which is detailed in Algorithm 5. The *j* loop (lines 2 $\sim$ 10) iterates through all coefficients in the polynomial; in each iteration, there is another *i* loop to accumulate intermediate results (lines 4 $\sim$ 9).

Figure 5a shows a simple example of parallel polynomial convolution on a GPU, where the polynomial length $N = 8$. Since each thread can perform the multiplication and accumulation independently, this technique is considered efficient for the GPU implementation. Recently, Lee *et al.* [15] proposed an improved version of this technique, wherein MAC operations are represented in a matrix form and computed entirely on tensor cores. However, it only support the polynomial convolution for lattice-based schemes that utilize small modulus, $q \leq 2^{11}$. Hence, it is not suitable to be used in Saber.

Referring to Table I, the modulus q of Saber parameter sets is always 8,192. Considering the Saber parameter set, coefficients of small polynomial (b) is in the range of -4 to $+4$, which can be conveniently represented in an 8-bit variable. Coefficients of polynomial (a), range from 0 to 8,191, can fit into a 16-bit variable. Referring to Figure 2, the DP2A instruction can be used to compute the dot-product operation between a pair of 16-bit/8-bit values. Hence, we can pack two 16-bit and two 8-bit coefficients into the respective 32-bit registers, then perform a series of dot-products to compute the polynomial convolution in Saber. Since each polynomial coefficient is packed into a 32-bit register with x and y components, loading these coefficients in parallel is a non-trivial task. In addition, the polynomial a is loaded in a cyclic

Algorithm 5 Nega-Cyclic Polynomial Convolution

```
1: procedure SCHOOLBOOK_POLY_CONV(out, a, b)
2:     for (j = 0; j < N; j++) do          ▷ N is the length of polynomial
3:         sum = 0;
4:         for (i = 0; i < j+1; i++) do          ▷ Accumulation
5:             sum = sum + a[j − i] × b[i];
6:         end for
7:         for (i = 1; i < N-j; i++) do          ▷ Subtraction
8:             sum = sum − a[j + i] × b[N − i];
9:         end for
10:        out[j] = sum;          ▷ out: array to store the results
11:    end for
12: end procedure
```

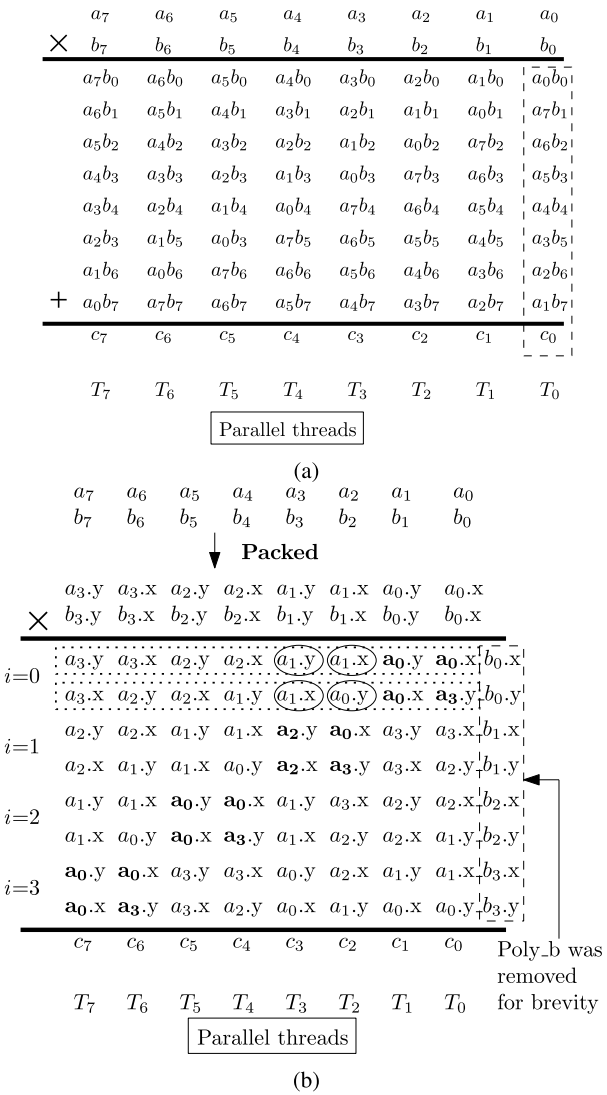

Fig. 5. Comparison of the proposed methods for polynomial convolution in Saber, (a) parallel implementation with int 32-bit integer units, (b) naïve implementation with DP2A instruction.

form, which is less straightforward compared to the case in a matrix-multiplication.

A detailed illustration of this problem is shown in Figure 5b. Considering thread 2 (T_2), it is reading the x component from

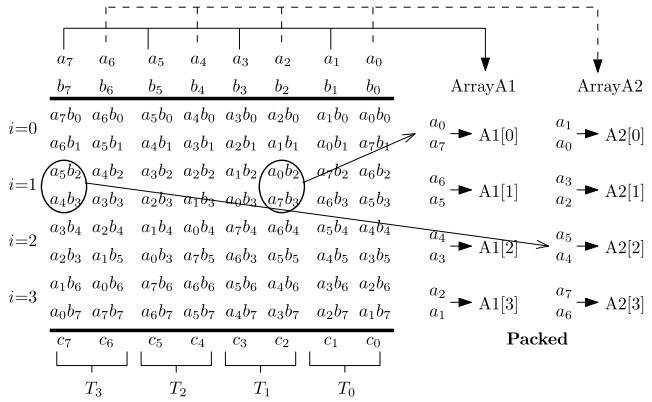

Fig. 6. Proposed packing method for polynomial convolution in Saber.

a_1 in the first iteration ($i = 0$), but it loads the y component from a_0 in the next iteration ($i = 1$). This inconsistency in the array index and component to be loaded exists in all even-index threads. Odd-index threads, i.e., $T_1, T_3, \ldots$, also need to access a different component from even-index threads. These access patterns cause many conditional statements in a naïve GPU implementation, since each thread needs to decide whether to read the x or y component and determine the index to read the polynomial a. This problem is not found in polynomial b, because it is always accessed in a sequential and fixed manner.

In this paper, we propose a novel technique to pack the polynomial a in order to avoid problems mentioned above. A closer look into Figure 5b reveals that all even-index threads are reading a different values, but they exhibit a cyclic pattern. For instance, when $i = 0, 1$, T_0 reads a_0.x and a_3.y; the same pair of values are being read by T_2, T_4, and T_6 at different time (See bolded parts). Same patterns also apply to odd-index threads. Due to this reason, we can pack the polynomial a into two arrays to cater for these two different accessing patterns. This is illustrated in Figure 6, in which the polynomial a is packed into two different arrays (ArrayA1 and ArrayA2) for parallel access. For instance, values a_0 and a_7 can be accessed in ArrayA1[0].x and ArrayA1[0].y; values a_5 and a_4 can be read from ArrayA2[2].x and ArrayA2[2].y.

Another important aspect in the implementation of parallel polynomial convolution is that Saber employs a nega-cyclic convolution, which needs to add or subtract the intermediate results when the i loop progresses. Referring to Figure 7, values to be added/subtracted are marked in black/blue colour respectively. A close look into this pattern reveals that values to be subtracted can be either the x or y component, depending on the thread index and index i. For instance, considering the case of odd-index threads, i.e., $T_1, T_3, \ldots$, they need to decide whether an addition or subtraction should be performed. On the other hand, considering the thread T_2, operations to be performed are different when $i = 0$, $i = 1$, and $i = 2, 3$. This happens to all other even-index threads, wherein there is always three different operations to be performed. This is because the value to be subtracted can be either stored on the x or y component of the values. This also implies that the implementation of parallel polynomial convolution in Saber is more complicated compared to FrodoKEM, as there are more conditional checking required.

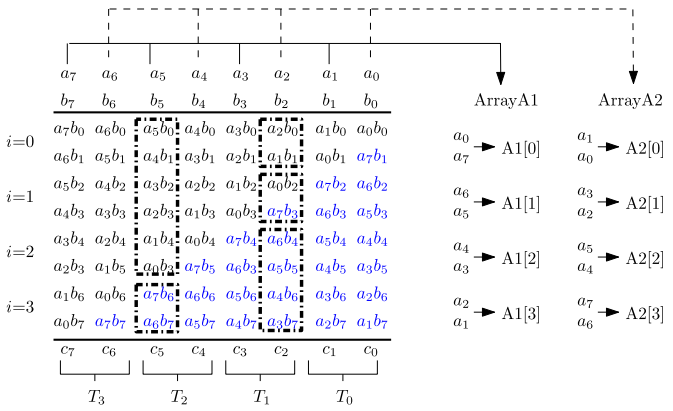

Fig. 7. Computing nega-cyclic polynomial convolution in Saber.

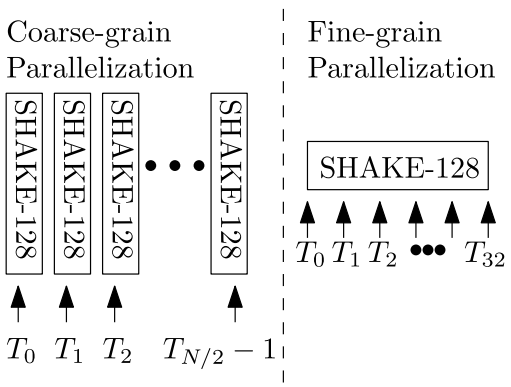

Fig. 8. Parallelization of the random samples generation.

Algorithm 6 DPSaber: Parallel Polynomial Convolution in Saber With Dot-Product Instruction

1: **procedure** POLY_CONV(out, $A1$, $A2$, b)
2: $sum1 = 0$, $sum2 = 0$;
3: _shared_ $s\_A1[tid] = A1[tid]$;
4: _shared_ $s\_A2[tid] = A2[tid]$;
5: _shared_ $s\_b[tid] = b[tid]$;
6: **for** $(i = 0; i < N/2; i++)$ **do** ▷ N is the length of polynomial
7: $load\_a = s\_A2[(tid \times (N/2-1)+i)\%(N/2)]$;
▷ Processing the even elements.
8: **if** $i > tid$ **then**
9: $sum1$ -= $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
10: **else if** $i == tid$ **then**
11: $sum1$ += $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
12: $sum1$ -= $load\_a.y \times s\_b[i].y + load\_a.y \times s\_b[i].y$;
13: **else**
14: $sum1$ += $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
15: **end if**
16: $load\_a = s\_A1[(tid + i \times (N/2-1))\%N/2]$;
▷ Processing the odd elements.
17: **if** $i \leq tid$ **then**
18: $sum2$ += $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
19: **else**
20: $sum2$ -= $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
21: **end if**
22: **end for**
23: out$[tid \times 2] = sum1$;
24: out$[tid \times 2 + 1] = sum2$;
25: **end procedure**

Detailed implementation steps are presented in Algorithm 6. Firstly, packed polynomials are loaded into shared memory to improve the accessing speed (lines 3 $\sim$ 5). Following this, the algorithm loads a value from s_A2 and proceeds to compute the dot-product operation (lines 8 $\sim$ 15). To process even-index threads, there are three different conditions, which should to be checked (lines 8, 10, and 13). This corresponds to problems explained in Figure 7 for even-index threads. The computation for odd-index threads is simpler as we only need to check the condition to perform an addition (line 17) and subtraction (line 20). Note that each thread computes one even and one odd element. This process is repeated for $N/2$ times (line 6) to complete the entire convolution. Finally, results of accumulations ($sum1$ and $sum2$) are stored into the output array following respective odd and even positions (lines 23 $\sim$ 24). With the proposed technique in Algorithm 6 the schoolbook polynomial convolution is mapped into a series of parallel dot-product operations. This allows efficient parallel polynomial convolution to be implemented on GPU, improving the speed performance against the conventional method that only uses 32-bit integer units.

IV. EXPERIMENTAL RESULTS

This section presents experimental results of FrodoKEM and Saber KEM on various GPU platforms, which are compared against the state-of-the-art works. Proposed algorithms are implemented in C language under CUDA 11.2 SDK. Performance was evaluated on two different platforms as described in Table II. Platform A is a workstation equipped with a Intel Core i9-10900K CPU and a RTX 3080 GPU. Platform B is the Compute Canada platform (a national computing grid) [32], which has a module configuration of four CPU cores (Xeon Gold), 16-GB RAM, and a GPU. The GPU can be configured as V100 or T4.

Our implementation offloads all the computations in encapsulation and decapsulation, including the pack/unpack process, random sample generations and hashing, onto the GPU platforms. Both FrodoKEM and Saber use SHAKE-128 [33] to generate random samples, but FrodoKEM uses much more random samples compared to Saber, so the implementation strategy is also different. The implementation of SHAKE-128 is illustrated in Figure 8. In our FrodoKEM implementation, we parallelized the SHAKE-128 algorithm in a coarse-grain manner. We launched K GPU blocks and $N/2$ threads, wherein each thread is performing its own SHAKE-128 to generate random samples independently. This strategy does not require any synchronization between threads, which is good for achieving high performance in GPU implementation. On the other hand,

TABLE II
EXPERIMENTAL PLATFORMS FOR PROPOSED IMPLEMENTATIONS

Platforms	CPU	Clock (GHz)	RAM (GB)	GPU	Architecture	No. Cores	Clock (GHz)	Mem. BW (GB/s)
A	i9-10900K	3.7	16	RTX3080	Ampere	8704	1.44	760.3
B	Xeon Gold 5120	2.2	16	V100	Volta	5120	1.25	897.0
				T4	Turing	2560	0.585	320.0

Algorithm 7 Encode Function in Saber

1: **procedure** POLT2BS($bytes$, $poly$)
2: **for** ($i = 0; i < N/2; i++$) **do**
3: $cnt_b = i$;
4: $cnt_p = 2 \times i$;
5: $bytes[cnt_b] = (poly[cnt_p]\&0x0f) | ((poly[cnt_p + 1]\&0x0f) \ll 4)$;
6: **end for**
7: **end procedure**

we do not need to generate a lot of random samples for Saber, so there may not be sufficient workload to fully exploit the GPU resources. Hence, we parallelized SHAKE-128 in a fine-grain manner in our Saber implementation, in which 32 threads cooperatively compute one SHAKE-128. This strategy requires synchronization between threads to avoid the potential data hazard [34], but it ensures that the GPU always has sufficient workload to compute, which is a very important aspect to ensure high performance on GPUs.

All other functions exhibit rich parallelism since they are operated on polynomial/matrix. In this case, we follow the fine-grain parallel approach, wherein one block consists of multiple threads are used to complete one operation. For instance, Algorithm 7 shows the encode function in Saber that converts all the coefficients in a polynomial to bytes. This operation is highly parallelizable, therefore we instantiate $N/2$ threads to parallelize the **for** loop in line 2. Since these functions are very lightweight, the fine-grain parallel approach ensures that there is sufficient workload to fully exploit the computational resources in a GPU.

We have selected FrodoKEM976 and Saber parameter sets for performance evaluation, since both belong to the NIST security category 3. To evaluate the throughput performance under different batch sizes, many parallel blocks are initiated, where the number of parallel blocks, K, varies. Each block is responsible in computing one KEM. In each experiment, K increases gradually to observe the achieved throughput, until the performance saturates. We have also compared our results with the AVX2 implementations of FrodoKEM and Saber on CPU clock at 3.7 GHz. The source code are obtained from the respective NIST submission package [4], [5] and compiled without any modification. Note that these AVX2 implementations are highly optimized by the authors of FrodoKEM and Saber. They are executed on a single CPU core throughout our experiments. The main focus of this paper is to show that we can exploit the dot-product feature in state-of-the-art GPU architectures to accelerate many lattice-based cryptographic schemes. However, dot-product instruction is not found in the existing x86 processors. Hence, we do not optimize the AVX2 implementation on multiple cores as this is not the focus of our work.

A. Evaluation of Proposed FrodoKEM Implementations (INT32 and DPFrodo)

Table III shows the throughput of computing one matrix multiplication in FrodoKEM using different implementation techniques. Results show that DPFrodo is at least 1.3× faster than the conventional implementation using integer unit, across different K on various GPU platforms. Performance saturates when K is relatively large ($\geq$512), indicating that further increasing the number of parallel blocks does not help in improving performance anymore.

The proposed DPFrodo technique is applied to FrodoKEM parameter set FrodoKEM976 to speed up the matrix multiplication. Referring to Table IV, DPFrodo is able to produce 1.09× higher throughput for both RTX 3080 and V100, and 1.07× for T4, against the conventional implementation utilizing 32-bit integer units. Compared to the implementation from Gupta *et al.* [12], which are considered state-of-the-art results, the proposed implementation is able to achieve 4.37× higher key exchange throughput on the same GPU (V100). The DPFrodo key exchange throughput is also higher than the AVX2 implementation by 6.59×, 3.65×, and 1.66×, on RTX3080, V100, and T4 GPU platforms, respectively.

B. Evaluation of Proposed Saber Implementations (INT32 and DPSaber)

In the Saber implementation, the polynomial convolution is used to perform two types of operations: inner product and matrix-vector multiplication. Table V shows throughput achieved in the proposed implementation. Considering the case of matrix-vector multiplication, when parallel blocks $K \geq 256$, DPSaber is able to achieve at least 1.13 higher throughput across all GPU platforms. The similar performance is also observed in the inner product, with exception of V100, wherein the throughput is only high enough when there $K \geq 1024$. Overall, the speed-up gain by DPSaber in the matrix-vector multiplication is more significant compared to inner product. This is because the proposed DPSaber requires some pre-computations to pack polynomials, which is a non-trivial overhead. Hence, the memory to compute ratio has to be large enough in order to capitalize benefits of dot-product instruction. The matrix-vector multiplication is performing more computations compared to the inner product, which explains why it can achieve a more significant speed-up even with a small K.

Table VI shows results of Saber KEM implementation on several GPUs across various block sizes (K), compared against the CPU AVX2 implementation. On RTX 3080 and V100,

TABLE III

COMPARISON OF PROPOSED MATRIX-MATRIX MULTIPLICATION IMPLEMENTATIONS IN FRODOKEM976 BASED ON 32-BIT INTEGER UNITS (INT32) AND DP2A (DPFRODO) INSTRUCTIONS

K	INT32	DPFrodo	Sp-up	INT32	DPFrodo	Sp-up	INT32	DPFrodo	Sp-up
	RTX 3080			**V100**			**T4**		
	Matrix-Matrix Multiplication (operations per second)								
64	44221	61159	1.38	32726	58062	1.77	14767	26907	1.98
128	173319	236334	1.36	123535	188501	1.53	29695	41712	1.40
256	178088	244308	1.37	150347	275649	1.83	48810	76973	1.58
512	182266	246275	1.35	157413	274175	1.74	59300	78916	1.33
1024	189441	245960	1.3	178323	294464	1.65	58801	78573	1.34
2048	193600	250790	1.3	180920	274288	1.52	60028	82634	1.38

TABLE IV

COMPARISON OF PROPOSED FRODOKEM976 SCHEME IMPLEMENTATIONS BASED ON INT 32-BIT INTEGER UNITS (INT32) AND DP2A (DPFRODO) INSTRUCTIONS

K	Encap Throughput			Decap Throughput			KX/s	
	INT32	DPFrodo	Sp-Up	INT32	DPFrodo	Sp-Up	INT32	DPFrodo
	RTX 3080							
64	7044	7356	1.04	7406	7804	1.05	3610	3610
128	8691	8994	1.03	9055	9229	1.02	4435	4555
256	9802	9912	1.01	10082	10554	1.05	5111	5111
512	10794	11247	1.04	10563	11261	1.07	5213	5627
768	11899	12769	1.07	11207	12083	1.08	**5771**	**6208**
	V100							
64	6108	6084	1	6559	6486	0.99	3163	3139
128	7305	7412	1.01	7098	7159	1.01	3600	3642
256	8544	8815	1.03	8249	8397	1.02	4197	4300
512	9388	9827	1.05	8771	8910	1.02	4535	4673
768	9624	10321	1.07	8920	9275	1.04	**4629**	**4885**
[12]	1749			1839			1117	
	T4							
64	2570	2562	1	4803	4720	0.98	1674	1661
128	3402	3396	1	4500	4491	1	1937	1934
256	5078	5161	1.02	4871	4957	1.02	2486	2528
512	5189	5398	1.04	4915	5160	1.05	**2534**	**2638**
768	5145	5390	1.05	4976	5153	1.04	2530	2634
	CPU, Intel Core i9-10900K (3700 MHz)							
AVX2 [1]	1428			1495			731	

[1] The source code was obtained from the FrodoKEM NIST submission package [4].

TABLE V

COMPARISON OF PROPOSED INNER PRODUCT AND MATRIX-VECTOR MULTIPLICATION IMPLEMENTATIONS IN SABER KEM BASED ON INT 32-BIT INTEGER UNITS (INT32) AND DP2A (DPSABER) INSTRUCTIONS

K	INT32	DPSaber	Sp-up	INT32	DPSaber	Sp-up	INT32	DPSaber	Sp-up
	RTX 3080			**V100**			**T4**		
	Matrix-Vector (thousands operations per second)								
64	500	530	1.06	406	381	0.94	169	177	1.05
128	862	992	1.15	710	702	0.99	193	273	1.42
256	1115	1645	1.48	1008	1136	1.13	236	338	1.43
512	1176	1931	1.64	1247	1475	1.18	258	391	1.52
1024	1185	1931	1.63	1278	1641	1.28	256	395	1.54
	Inner Product (thousands operations per second)								
64	3289	1533	0.47	1838	962	0.52	574	505	0.88
128	3799	2551	0.67	2976	1812	0.61	617	788	1.28
256	4262	4635	1.09	3731	2874	0.77	718	982	1.37
512	4000	5646	1.41	4903	4276	0.87	794	1157	1.46
1024	4441	6491	1.46	5906	6038	1.02	781	1186	1.52

DPSaber is able to achieve higher key exchange (KX/s) rate compared to the conventional implementation using 32-bit integer (INT32), when parallel blocks (K) are more than 256. On T4, DPSaber is better than INT32 for all cases starting from $K = 64$. The best result was achieved when the $K = 768$ or $K = 512$, where DPSaber is 1.09×, 1.02×, and 1.19× faster than INT32 implementation on RTX 3080, V100, and T4 GPUs, respectively. The key exchange throughput achieved by DPSaber implementation is 4.17×, 4.04×, and 1.06× higher than AVX2 implementation on RTX 3080, V100, and T4, respectively.

The performance of our FrodoKEM and Saber implementation on various GPUs differs, which is mainly due to the differences in clock frequency, memory bandwidth and the

TABLE VI
COMPARISON OF PROPOSED SABER KEM IMPLEMENTATIONS BASED ON INT 32-BIT INTEGER UNITS (INT32) AND DP2A (DPSABER) INSTRUCTIONS

K	Encap Throughput			Decap Throughput			KX/s	
	INT32	DPSaber	Sp-Up	INT32	DPSaber	Sp-Up	INT32	DPSaber
RTX 3080								
64	104178	101906	0.98	105535	98980	0.94	52426	50211
128	163172	163995	1.01	154595	153421	0.99	79384	79266
256	206276	220580	1.07	191856	204405	1.07	99402	106092
512	231094	251707	1.09	209578	229542	1.1	109905	120057
768	238367	262209	1.1	219714	236761	1.08	**114330**	**124418**
V100								
64	27354	26974	0.99	17468	17134	0.98	10660	10478
128	49748	49839	1	58417	55599	0.95	26868	26281
256	80802	81494	1.01	155461	155751	1	53168	53501
512	115921	118909	1.03	182066	181402	1	70826	71827
768	187361	194557	1.04	298764	316310	1.06	**117827**	**120463**
T4								
64	15548	15620	1	37383	37218	1	10981	11002
128	25749	32596	1.27	48744	53288	1.09	16849	20225
256	37666	48195	1.28	56559	64750	1.14	22609	27630
512	47707	58157	1.22	60700	69480	1.14	**26712**	**31658**
768	51888	62400	1.2	61052	70585	1.16	16849	20225
CPU, Intel Core i9-10900K (3700 MHz)								
AVX2 [1]	52835			53981			29836	

[1] The source code was obtained from the Saber NIST submission package [5].

number of cores. Referring to Table II, RTX 3080 has a higher number of cores and clock frequency among all GPUs, so it is the best achieving candidate in our experiments. Despite having lesser number of cores, the performance of V100 is not far from RTX 3080, because it has a slightly higher memory bandwidth. This also means that V100 can deliver more data for computation compared to RTX 3080, which can be an advantage. In GPU architecture, the warp scheduler may stall the computation while it is waiting for data from the memory. Hence, a higher memory bandwidth can supply sufficient data for computation and reduce the chances of stalls, effectively offsetting the disadvantages of having lower number of cores. The performance in T4 is much slower than the other two GPUs due to the lowest rank on all these aspects. Although these three GPUs have a different architecture, it does not seriously affect their performance in our experiments. For instance, the matrix multiplication and polynomial convolution heavily rely on the INT32 unit and dot-product instructions, which has the same throughput for Turing, Volta and Ampere architecture [19]. Similarly, these three architectures also have the same number of warp schedulers (four) [19], so the instruction scheduling performance is almost similar. In conclusion, the architectural differences may not be the main reason for the performance differences in our implementation.

C. Bottleneck Analysis of the Proposed DPFrodo and DPSaber

The DP2A instruction can perform two multiplications and one accumulate operation in a clock cycle. Theoretically, the speed up that we can gain from this should be $2\times$ faster than using the conventional 32-bit integer. However, there are several factors that limits this gain, which is discussed in this subsection.

1) Matrix-Matrix Multiplication in FrodoKEM: The packing and ordering of data into the format required by DP2A instruction causes non-negligible overhead. For the case in FrodoKEM, matrix A was originally stored in a column-major format. To perform matrix-matrix multiplication, A is accessed in a row-major manner, which requires additional effort in packing two matrix elements from different memory locations (refer to Figure 4). Moreover, the even-indexed threads are accessing different memory locations compared with the odd-indexed threads, causing a warp divergence effect. There are two possible ways to eliminate these undesirable effects, which we have experimentally verified to be inefficient.

1) Method 1: One can try to reorganize matrix A in a form that is friendly to matrix-matrix multiplication (row-major), removing the need to pack the matrix elements for DP2A instruction. However, due to its large size, rearranging matrix A is also a non-trivial task, eventually harming the performance.
2) Method 2: Alternatively, one can also modify Algorithm 4 to first perform the operations for all even-indexed threads, then proceed to complete the operations for odd-indexed threads. The steps are demonstrated in Algorithm 8. This eliminates the warp divergence issue, but it introduces significant impact in performance. The number of parallel threads are reduced by half, since the GPU kernel is computing either only the even-indexed or odd-indexed part. This reduces the parallelism achieved and affected the overall performance. Besides, the computed values are stored to the output array (lines 13 $\sim$ 14 and lines 25 $\sim$ 26) twice in a non-coalesced manner.

Referring to Table VII, the matrix reordering in Method 1 takes up too much time, eventually nullifying all the benefits from removing warp divergence. Similarly, Method 2 eliminated the warp divergence effect through Algorithm 8, but the performance is still slower than the proposed Algorithm 4. Through these analysis, we believe that although Algorithm 4 exhibits warp divergence, it is still better than other techniques

Algorithm 8 Method 2: Warp Divergence Free Parallel Matrix Multiplication in FrodoKEM With Dot-Product Instruction

1: **procedure** MAT_MUL_AND_ADD_UNROLL(out, A, s)
2: $sum0 = 0$, $sum1 = 0$, $sum2 = 0$, $sum3 = 0$;
3: $sum4 = 0$, $sum5 = 0$, $sum6 = 0$, $sum7 = 0$;
4: Process even-indexed threads
5: **for** ($j = 0$; $j < N/2$; $j++$) **do**
6: $load\_a.x = A[j * N + tid/2].x$;
7: $load\_a.y = A[j * N + N/2 + tid/2].x$;
8: $sum0 \mathrel{+}= load\_a.x \times s[j].x + load\_a.y \times s[j].y$;
9: $sum1 \mathrel{+}= load\_a.x \times s[1 \times N + j].x +$
10: $load\_a.y \times s[1 \times N + j].y$;
11: … ▷ Unroll 8 times (*N'*). Removed for brevity
12: **end for**
13: out$[2 * tid] = sum0$;
14: out$[1 \times N + 2 * tid] = sum1$;
15: … ▷ Unroll 8 times (*N'*). Removed for brevity
16: Process odd-indexed threads
17: **for** ($j = 0$; $j < N/2$; $j++$) **do**
18: $load\_a.x = A[j * N + tid/2].y$;
19: $load\_a.y = A[j * N + N/2 + tid/2].y$;
20: $sum0 \mathrel{+}= load\_a.x \times s[j].x + load\_a.y \times s[j].y$;
21: $sum1 \mathrel{+}= load\_a.x \times s[1 \times N + j].x +$
22: $load\_a.y \times s[1 \times N + j].y$;
23: … ▷ Unroll 8 times (*N'*). Removed for brevity
24: **end for**
25: out$[2 * tid + 1] = sum0$;
26: out$[1 \times N + 2 * tid + 1] = sum1$;
27: … ▷ Unroll 8 times (*N'*). Removed for brevity
28: **end procedure**

TABLE VII
ANALYZING THE BOTTLENECK IN DPFRODO

	RTX 3080						
K	INT32	DPFrodo	Sp-up	Met. 1	Sp-up	Met. 2	Sp-up
	Matrix-Matrix Multiplication (operations per second)						
256	178088	244308	1.37	130847	0.73	155189	0.87

to implement FrodoKEM matrix-matrix multiplication using DP2A instruction.

2) Matrix-Vector and Inner Product in Saber: Considering the case in DPSaber, the main bottleneck of polynomial convolution lies in the multiple conditional statements (Algorithm 6, lines 8, 10, 13, 17 and 19). This introduces warp divergence issue on GPU execution and eventually limited the performance gain. However, these conditional statements are unavoidable because each thread needs to select a different components (x or y) from the packed polynomial and decide to add or subtract from the intermediate results. This implies that the computational pattern is slightly different for each thread when it is executed in parallel, mainly caused by the nega-cyclic convolution used in Saber. Unlike the case in DPFrodo, we are unaware of any trivial methods to avoid the warp divergence. Despite the warp divergence issue, DPSaber can still benefit from the dot-product instruction if the number of parallel blocks (K) is sufficiently large.

3) Performance Breakdown and Discussions: Table VIII shows the breakdown of the main computations in FrodoKEM and Saber when they are implemented on CPU and GPU platforms. We can observe that majority of the computations in FrodoKEM (for both GPU and CPU) are attributed to the random samples generation, which is done through SHAKE-128 algorithm. After extensive optimization using the proposed dot-product technique, matrix multiplication is no longer the main bottleneck. Optimizing the serial algorithm like SHAKE-128 can significantly improve the throughput of FrodoKEM. An alternative way is to employ a variant of FrodoKEM using AES [4] to generate random samples. Many existing works that optimized the implementation of AES on GPUs [10] can be leveraged to speed up the random sample generation in FrodoKEM, but this is out of the scope of this paper.

Similar to FrodoKEM, random sample generation is also the main bottleneck (≈40%) in Saber, which is also performed through SHAKE-128. This implies that optimizing SHAKE-128 implementation on GPUs can also benefit Saber. After heavy optimization using the proposed dot-product technique, the proportion of matrix-vector multiplication and inner product are reduced. However, they are still consuming noticeable time in Saber. Unlike FrodoKEM, hash operations contribute to a significant part of the computation in Saber. However, the hash function used in Saber (SHA3-256) is very similar to SHAKE-128; optimizing the implementation performance of SHAKE-128 is likely to benefit SHA3-256 as well.

D. Applying the Dot-Product Solutions to Toom-Cook and NTT

The technique proposed in Algorithm 6 can be applied to other asymptotically faster algorithms like Karatsuba [35] and k-way Toom-Cook [36]. These algorithms decomposed the polynomial with length N into sub-polynomials with a smaller length. This decomposition can be applied recursively to obtain a smaller sub-polynomials. Karatsuba is a special case in k-way Toom-Cook, where $k = 2$. As a concrete example, the Saber [5] submission package to NIST uses 4-way Toom-Cook to speed up the polynomial convolution. The polynomial ($N = 256$) is first decomposed into four equally sized sub-polynomials ($N = 64$), and then perform the evaluation, multiplication and interpolation on these small sub-polynomials. The multiplication stage involves sub-polynomials with $N = 64$, which can be computed using the proposed dot-product-aided schoolbook multiplication (Algorithm 6). In this way, we believe that the proposed technique can provide a better performance. Note that one can also continue to apply the 4-way Toom-Cook or Karatsuba algorithm recursively. However, the level of parellelism is reduced by half for each level of recursion, which may eventually harm the implementation performance on a GPU platform. Recall that the smallest instruction scheduling unit on GPU is a warp (32 threads), it is advisable to have $N \geq 32$ at the last recursive level. Note that it is non-trivial to explore the best combination of Toom-Cook, Karatsuba and schoolbook multiplication to achieve the highest throughput. Hence, we intend to leave this exploration as a future research direction.

On the other hand, many lattice-based PQC schemes utilize the NTT to compute the polynomial convolution on a special

TABLE VIII
PERFORMANCE BREAKDOWN OF FRODOKEM AND SABER IMPLEMENTED ON GPUS AND CPU

DPFrodo: dot-product aided FrodoKEM		RTX 3080	V100	T4	CPU (AVX2)
Encapsulation	Mem. copy between CPU and GPU	2%	1%	1 %	–
	Generating random samples	83%	86%	76%	68%
	Matrix multiplications	7%	4%	7%	17%
	Hash operations	5%	5%	11%	10%
	Others (pack/unpack, add, encode, CDF, etc.)	6%	4%	4%	5%
Decapsulation	Mem. copy between CPU and GPU	4%	4%	4%	–
	Generating random samples	78%	73%	72%	66%
	Matrix multiplications	6%	4%	7%	16%
	Hash operations	4%	3%	7%	10%
	Others (pack/unpack, add, encode, CDF, etc.)	8%	16%	6%	8%
DPSaber: dot-product aided Saber		RTX 3080	V100	T4	CPU (AVX2)
Encapsulation	Mem. copy between CPU and GPU	7%	5%	3%	–
	Generating random samples	42%	37%	47%	49%
	Matrix-vector / Inner product	17%	20%	21%	30%
	Hash operations	33%	36%	26%	16%
	Others (pack/unpack, add, encode, CDF, etc.)	1%	1%	3%	5%
Decapsulation	Mem. copy between CPU and GPU	22%	17%	10%	–
	Generating random samples	39%	39%	40%	46%
	Matrix-vector / Inner product	19%	21%	30%	32%
	Hash operations	18%	20%	18%	15%
	Others (pack/unpack, add, encode, CDF, etc.)	2%	3%	2%	7%

Algorithm 9 Pseudocode: Fully Parallel in-Place NTT in Kyber [13]

```
1: procedure NTT(r)
2:     r : Polynomial with length N = 256
3:     zeta and zeta_tb : Precomputed NTT constants
4:     for len=128; len≥ 2; len=len/2 do
5:         zeta = zeta_tb[level + ⌊tid/len⌋];
6:         j = ⌊tid/len⌋ * len + tid;
7:         t = fqmul(zeta, r[j + len]);
8:         r[j + len] = r[j] - t;
9:         r[j] = r[j] + t;
10:        level = level × 2;
11:    end for
12: end procedure
```

ring structure. Some representative candidates include Kyber KEM [2] and Dilithium signature [3]. However, it is non-trivial to compute the NTT using dot-product instruction. Algorithm 9 shows an example of the parallel implementation of NTT in Kyber, which is proposed by Lee *et al.* [13]. The core computation in NTT boils down to the butterfly operations (lines 8 $\sim$ 10), which usually involves the multiplication between an input and a constant, e.g., twiddle factor, followed by a modular operation over the modulus q. This corresponds to line 8 in Algorithm 9, where the twiddle factor (*zeta*) is multiplied with the input from polynomial r and reduced by modulus q. Note that this modular operation is usually performed through Montgomery or Barrett reduction algorithm, which is not easily mapped to a dot-product operation. This is also why the previous implementations of Kyber on various platforms [37] do not consider using a dot-product instruction to speed up the performance.

E. Practical Use Cases of the Proposed GPU Implementation

1) Secure Online Transactions: Key exchange is a fundamental feature supported by many security protocols such as SSL/TLS [38] and IPsec (IKE) [39]. KEM can be used to support the key exchange between the client/server for Internet communication. Under such communication paradigm, the server is required to process massive amount of KEM, i.e., hundreds of thousands, requests from various clients within a short period of time. This situation is especially common for e-commerce, online banking, and transactions. It is challenging to cope with such a demanding and ever increasing computations, even for a very high performance server, as the server itself may need to handle other computations as well. One of the possible solutions is to offload the KEM computations to hardware accelerators like FPGA and GPU, which are more specialized in performing batch computation. Proposed high throughput implementation on various GPU platforms shows that it is possible to perform thousands to hundreds of thousands key-encapsulation/decapsulations per second. By utilizing the proposed solution, we can effectively offload batch computations of KEMs to GPU, eventually save a lot of time in CPU, which then allows the server to execute some other tasks. Note that GPUs are already commonly found in many major cloud computing services like AWS [40] and IBM [41], as GPUs are widely used for artificial intelligence. Hence, the use of GPU in accelerating KEMs is a more natural choice compared to FPGA or ASIC solution.

2) Secure Internet of Things Communication: Another interesting use case is the IoT applications, wherein the sensor nodes are actively interacting with the cloud servers. The scale of IoT system range from hundreds to thousands of sensor nodes [42]. To secure such communication, symmetric keys used to encrypt the sensor data, need to be refreshed frequently, which can be done through one of these ways:

1) New session keys are produced by the IoT sensor and transmitted to the cloud server via KEM. Typically, the symmetric key is refreshed in every communication session using pseudorandom number generator or KDF.
2) The cloud server produced many new session keys and send them to each sensor node via KEM for update.

In such a case, the cloud server 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 requires the cloud server to decapsulate and obtain session keys, while the second way requires the cloud server to encapsulate many session keys. Regardless of the chosen method, the cloud server needs to perform a lot of KEM computations in a timely manner. Hence, a high throughput KEM proposed in this paper can be very useful to offload compute-intensive KEM computations on GPU, leaving the cloud server with more resources to handle the other important computations.

V. Conclusion

In this paper, we are the first to show that the dot-product instruction (DP2A) offered by modern NVIDIA GPU architectures can be used to accelerate lattice-based cryptographic schemes. A highly optimized implementation of matrix-matrix multiplication is presented, which allows the proposed FrodoKEM implementation to be 4.37× faster than the state-of-the-art work proposed by Gupta *et al.* [12]. A novel data structure is also proposed to enable the efficient computation of the parallel polynomial convolution by using the DP2A instruction. Note that these two proposed techniques are generic, i.e., they can be adapted to any parallel processor architectures that offer dot-product instructions. For instance, the latest AMD GPU also supports similar dot-product instructions, e.g., V_DOT2_U32_U16 [43], which is a good candidate to adopt the proposed method to speed-up lattice-based cryptographic schemes. Moreover, the proposed technique can be used for LAC [44], which is a NIST round 2 candidate. A more optimized implementation of random sample generation (through AES or SHAKE) can also help in improving the performance of lattice-based cryptographic schemes on GPU platforms.

References

[1] *Post-Quantum Cryptography: Round 3 Submissions*. Accessed: Jul. 18, 2021. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

[2] J. Bos *et al.*, "CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Apr. 2018, pp. 353–367.

[3] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium: A lattice-based digital signature scheme," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, no. 1, pp. 238–268, Nov. 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/839/791

[4] E. Alkim *et al.* (2020). *FrodoKEM learning with errors key encapsulation*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/FrodoKEM-Round3.zip

[5] J.-P. D'Anvers *et al.* (2020). *Saber: Mod-LWR based KEM*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/SABER-Round3.zip

[6] S. Dong *et al.*, "Spartan: A sparsity-adaptive framework to accelerate deep neural network training on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 10, pp. 2448–2463, Oct. 2021.

[7] N. Tahmasebi, P. Boulanger, J. Yun, G. Fallone, M. Noga, and K. Punithakumar, "Real-time lung tumor tracking using a CUDA enabled nonrigid registration algorithm for MRI," *IEEE J. Translational Eng. Health Med.*, vol. 8, pp. 1–8, 2020.

[8] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 2, pp. 941–956, Apr. 2019.

[9] C. Tezcan, "Optimization of advanced encryption standard on graphics processing units," *IEEE Access*, vol. 9, pp. 67315–67326, 2021.

[10] W.-K. Lee, H. J. Seo, S. C. Seo, and S. O. Hwang, "Efficient implementation of AES-CTR and AES-ECB on GPUs with applications for high-speed FrodoKEM and exhaustive key search," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, early access, Apr. 4, 2022, doi: 10.1109/TCSII.2022.3164089.

[11] S. Sun, R. Zhang, and H. Ma, "Efficient parallelism of post-quantum signature scheme SPHINCS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2542–2555, Nov. 2020.

[12] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: FrodoKEM, NewHope, and kyber," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 575–586, Mar. 2021.

[13] W. K. Lee and S. O. Hwang, "High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of Things applications," *IEEE Trans. Services Comput.*, early access, Aug. 10, 2021, doi: 10.1109/TSC.2021.3103956.

[14] Y. Gao, J. Xu, and H. Wang, "CuNH: Efficient GPU implementations of post-quantum KEM NewHope," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 551–568, Mar. 2021.

[15] W.-K. Lee, H. Seo, Z. Zhang, and S. Hwang. (2021). *Tensorcrypto*. Cryptology ePrint Archive, Report 2021/173. [Online]. Available: https://ia.cr/2021/173

[16] A. Biswas and A. P. Chandrakasan, "CONV-SRAM: An energy-efficient SRAM with in-memory dot-product computation for low-power convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 217–230, Jan. 2018.

[17] J. Vreca *et al.*, "Accelerating deep learning inference in constrained embedded devices using hardware loops and a dot product unit," *IEEE Access*, vol. 8, pp. 165913–165926, 2020.

[18] *A64—SVE Instructions*. [Online]. Available: https://developer.arm.com/documentation/ddi0596/2020-12/SVE-Instructions?lang=en

[19] *CUDA C Programming Guide, Version 11.6*, NVIDIA Corp., Santa Clara, CA, USA, 2022.

[20] O. Regev, "New lattice-based cryptographic constructions," *J. ACM*, vols. 6–51, pp. 899–942, Nov. 2004, doi: 10.1145/1039488.1039490.

[21] J. Bos *et al.*, "Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1006–1018.

[22] J. W. Bos, M. Ofner, J. Renes, T. Schneider, and C. V. Vredendaal, "The matrix reloaded: Multiplication strategies in FrodoKEM," in *Proc. Int. Conf. Cryptol. Netw. Secur.* Springer, 2021, pp. 72–91.

[23] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Advances in Cryptology*. 2012, pp. 719–737, doi: 10.1007/978-3-642-29011-4_42.

[24] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. *PQM4: Post-Quantum Crypto Library for the ARM Cortex-M4*. Accessed: Aug. 31, 2021. [Online]. Available: https://github.com/mupq/pqm4

[25] H. Becker, J. M. B. Mera, A. Karmakar, J. Yiu, and I. Verbauwhede, "Polynomial multiplication on embedded vector architectures," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2022, no. 1, pp. 482–505, Nov. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9305

[26] J. M. B. Mera, A. Karmakar, and I. Verbauwhede, "Time-memory trade-off in Toom–Cook multiplication: An application to module-lattice based cryptography," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 2, pp. 222–244, Mar. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8550

[27] J. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," in *Advances in Cryptology* (Lecture Notes in Computer Science), vol. 10831, A. Joux, A. Nitaj, and T. Rachidi, Eds. Marrakesh, Morocco: Springer, May 2018, pp. 282–305, doi: 10.1007/978-3-319-89339-6_16.

[28] H. Seo, P. Sanal, A. Jalali, and R. Azarderakhsh, "Optimized implementation of SIKE round 2 on 64-bit ARM cortex—A processors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 8, pp. 2659–2671, Aug. 2020.

[29] Y. Zhu *et al.*, "LWRpro: An energy-efficient configurable crypto-processor for module-LWR," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 3, pp. 1146–1159, Mar. 2021.

[30] J. M. Pollard, "The fast Fourier transform in a finite field" *Math. Comput.*, vol. 25, no. 114, pp. 365–374, 1971. [Online]. Available: http://www.jstor.org/stable/2004932
[31] A. Karmakar, J. M. B. Mera, S. S. Roy, and I. Verbauwhede, "Saber on ARM: CCA-secure module lattice-based key encapsulation on ARM," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, no. 3, pp. 243–266, Aug. 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7275
[32] (2021). *Compute Canada*. Accessed: Oct. 10, 2021. [Online]. Available: https://www.computecanada.ca/home/
[33] P. Pritzker and P. D. Gallagher, "SHA-3 standard: Permutation-based hash and extendable-output functions," Inf. Tech Lab. Nat. Inst. Standards Technol., Tech. Rep., 2014, pp. 1–35. Accessed: Nov. 1, 2021. [Online]. Available: https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions
[34] W.-K. Lee, X.-F. Wong, B.-M. Goi, and R. C.-W. Phan, "CUDA-SSL: SSL/TLS accelerated by GPU," in *Proc. Int. Carnahan Conf. Secur. Technol. (ICCST)*, Oct. 2017, pp. 1–6.
[35] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Dokl. Akad. Nauk SSSR* vol. 145, no. 2, pp. 293–294, Feb. 1962.
[36] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Math. Doklady*, vol. 3, no. 4, pp. 714–716, 1963.
[37] E. Alkim, Y. A. Bilgin, M. Cenk, and F. Gérard, "Cortex-M4 optimizations for R,M LWE schemes," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 3, pp. 336–357, Jun. 2020.
[38] E. Rescorla and T. Dierks, "The transport layer security (TLS) protocol 947 version 1.3," Tech. Rep., 2018. Accessed: Feb. 1, 2022. [Online]. Available: https://tools.ietf.org/id/draft-ietf-tls-tls13-23.html
[39] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, "Internet key 949 exchange protocol version 2 (IKEv2)," RFC 5996, Tech. Rep., Sep. 2010. Accessed: Feb. 1, 2022. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5996
[40] (2021). *Amazon EC2 P4d Instances*. Accessed: Oct. 10, 2021. [Online]. Available: https://aws.amazon.com/ec2/instance-types/p4/
[41] (2021). *NVIDIA GPUs on IBM Cloud Servers*. Accessed: Oct. 10, 2021. [Online]. Available: https://www.ibm.com/cloud/gpu
[42] F. Cirillo, D. Gomez, L. Diez, I. E. Maestro, T. B. J. Gilbert, and R. Akhavan, "Smart city IoT services creation through large-scale collaboration," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 5267–5275, Jun. 2020.
[43] *RDNA 2 Instruction Set Architecture, Reference Guide, Advanced Micro Devices*. Accessed: Sep. 30, 2021. [Online]. Available: https://developer.amd.com/wp-content/resources/RDNA2_Shader_ISA_Novembe%r2020.pdf
[44] X. Lu *et al.*, "LAC: Practical Ring-LWE based public-key encryption with byte-level modulus," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 1009, 2018.

Wai-Kong Lee (Member, IEEE) received the B.Eng. and M.Sc. degrees in electronics from Multimedia University in 2006 and 2009, respectively, and the Ph.D. degree in engineering from the Universiti Tunku Abdul Rahman, Malaysia, in 2018. He is currently a Post-Doctoral Researcher with Gachon University, South Korea. His research interests are in the areas of cryptography, numerical algorithms, GPU computing, the Internet of Things, and energy harvesting.

Hwajeong Seo (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Pusan National University. He is currently an Assistant Professor with Hansung University. His research interest includes cryptographic engineering.

Seong Oun Hwang (Senior Member, IEEE) 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, in 2004. From 1994 to 1996, he worked as a Software Engineer with LG-CNS Systems Inc. From 1998 to 2007, he worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI). 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. His research interests include cryptography, cybersecurity, and artificial intelligence. He is an Editor of *ETRI Journal*.

Ramachandra Achar (Fellow, IEEE) received the B.Eng. degree in electronics engineering from Bangalore University, India, in 1990, the M.Eng. degree in micro-electronics from the Birla Institute of Technology and Science, Pilani, India, in 1992, and the Ph.D. degree in electrical engineering from Carleton University in 1998. He is currently a Professor with the Department of Electronics Engineering, Carleton University. Prior to joining Carleton University Faculty in 2000, he worked in various capacities in leading research laboratories, including the T. J. Watson Research Center, IBM, New York, in 1995, Larsen and Toubro Engineers Ltd., Mysore, in 1992, the Central Electronics Engineering Research Institute, Pilani, India, in 1992, and the Indian Institute of Science, Bengaluru, India, in 1990. His research interests include signal/power integrity analysis, circuit simulation, parallel and numerical algorithms, EMC/EMI analysis, and mixed-domain analysis. He is a fellow of the Engineers Institute of Canada. He is a Practicing Professional Engineer of Ontario.

Angshuman Karmakar received the B.E. degree in computer science and engineering from Jadavpur University, Kolkata, the M.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, and the Ph.D. degree from Katholieke Universiteit Leuven (KU Leuven), Belgium, for his dissertation titled "Design and Implementation Aspects of Post-Quantum Cryptography." He is one of the primary designers of the post-quantum Saber KEM scheme which is one of the finalists in the NIST's post-quantum standardization procedure. He is currently an FWO Post-Doctoral Fellow with the COSIC Research Group, KU Leuven. His research interest spans different aspects of lattice-based post-quantum cryptography and computation on encrypted data.

Jose Maria Bermudo Mera received the B.Eng. and M.Sc. degrees in telecommunications engineering from the Technical University of Madrid, Spain. He is currently pursuing the Ph.D. degree with the COSIC Research Group, Katholieke Universiteit Leuven, Belgium, with a research project titled "Implementation Aspects of Lattice-Based Cryptography." He is a Team Member of the post-quantum Saber key-encapsulation mechanism scheme which is one of the finalists in the NIST's post-quantum standardization procedure. His research interests include the implementation of cryptography on software and hardware platforms and physical security.