IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–20. DOI:XXXXXXXX

DPCrypto: Acceleration of Post-quantum Cryptographic Algorithms using Dot-Product Instruction on GPUs

Wai-Kong Lee[1], Hwajeong Seo[2], Seong Oun Hwang[1], Angshuman Karmakar[3], Jose Maria Bermudo Mera[3] and Ramachandra Achar[4]

[1] Gachon University, Seongnam, South Korea, waikong.lee@gmail.com,sohwang@gachon.ac.kr
[2] Hansung University, Seoul, South Korea.
[3] KU Leuven, Leuven, Belgium.
[4] Carleton University, Ottawa, Canada.

Abstract. Dot-product is a widely used operation in many machine learning and scientific computing algorithms. Recently, NVIDIA has introduced dot-product instructions (DP2A and DP4A) in modern GPU architectures, 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 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 commonly found 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 in terms of key exchange operations per second than the state-of-the-art implementation on V100 GPU. This paper also presents the first implementation of Saber on GPU platforms, achieving 124,418, 120,463, and 31,658 key exchange operations 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, our proposed techniques are likely to benefit other similar algorithms. The proposed high throughput implementation of KEMs on various GPU platforms allows the heavy computations (KEMs) to be offloaded from the server. This is very useful for many emerging applications like Internet of Things and cloud computing.

Keywords: Post-quantum Cryptography · Dot-product · Polynomial Convolution · Matrix-multiplication · Graphics Processing Unit · FrodoKEM · Saber

1 Introduction

In 2016, National Institute of Standards and Technology (NIST) [NIS17] initiated a standardization process to select key encapsulation mechanism (KEM), public key encryption, and digital signature schemes that are resistant to quantum computer attacks. This is a timely response to the threat from quantum computers that can break existing RSA and elliptic-curve discrete logarithm based public key cryptography schemes. This standardization process has stimulated a lot of interest in post-quantum cryptography (PQC), which focuses on improving the security of PQC algorithms and performance of their implementations. Currently, the standardization is in Round-3, wherein 15 candidates are selected [NIS]. Among these 15 Round-3 candidates, seven of them are based on lattice hard problems. One of the main performance bottlenecks in lattice-based cryptography is polynomial convolution or matrix multiplication. Some schemes (e.g., Kyber and Dilithium) are based on a special ring structure that allows 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 and Saber), require carefully designed implementations in order to achieve reasonably fast performance.

Graphics Processing Unit (GPU) with a massively parallel architecture can be used in accelerating non-graphics computation, which de-facto makes it one of the accelerators available in many server-side environments. GPU has been widely used to speed-up algorithms in various domains, including deep learning [DSA$^+$21], power engineering [SHLW19], and healthcare [TBY$^+$20]. Recently, there are also several attempts on utilizing GPU for implementing cryptographic algorithms. For instance, an attempt to accelerate homomorphic encryption using GPU was presented by Al Badawi et al. [ABPA$^+$19, ABVMA18], and later on extended to support multiple GPUs [ABVL$^+$20]. Another similar work that utilized GPU for implementing high performance homomorphic encryption was presented by Lei et al. [LGZ$^+$19]. Besides that, GPU was also used to implement symmetric key cryptographic algorithms [LGP19, HMKG19, Tez21] achieving high throughput.

Since the commencement of NIST standardization process, there are some research works that explore the possibility of accelerating PQC with GPU. One notable work was presented by Sun et al., wherein they exploited the parallel architecture in GPU to implement the tree structure in SPHINCS signature scheme [SZM20]. Another attempt along this direction is the work by Akleylek et al. [ASL$^+$20]. They proposed a novel PQC scheme based on multivariate quadratic problems and instantiate a practical implementation using GPU. However, these two schemes are not included in the Round-3 of NIST standardization. Gupta et al. [GJCC21] presented a comprehensive benchmark of FrodoKEM, NewHope, and Kyber on various GPU platforms. Authors proposed *single* mode and *batch* mode to compute PQC algorithms on GPU. Recently, Lee et al. [LH21] and Gao et al. [GXW21] also showcased high throughput implementations of Kyber and NewHope KEM on GPU. 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 advanced features found in modern GPU architectures.

Since 2017, NVIDIA has released several modern GPU architectures that come with special purpose computing units. Tensor core was firstly introduced in the Volta architecture to speed-up the computations in machine learning algorithms. Although it is intended for machine learning applications, the introduction of tensor core in GPU is also beneficial to other domains [NCB$^+$20]. In a recent work, Lee et al. has introduced some techniques to compute polynomial convolution and matrix-multiplication using tensor core in GPU [LSZH21]. The key idea is to pack many polynomials into a matrix form and compute them efficiently using the tensor core in a GPU. Although performance improvements are impressive, this technique can only achieve its full benefit if the usage of non-ephemeral key is permitted. Moreover, this technique relies on fast tensor core that supports half precision, which implies that it cannot be used by lattice-based cryptography schemes that has a large modulus ($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 intend to fill this research gap by proposing novel implementation techniques on GPU that can be applicable to a larger modulus size.

1.1 Motivations

Dot-product is a common operation found in various algorithms. Due to this reason, many research works have been devoted to design specific hardware for dot-product computation

[BC18, VSG$^+$20], which offers faster and more energy efficient solutions than straightforward implementations. The popular processor architecture ARM has released special instructions [ARM] to handle dot-product operation. Unsurprisingly, NVIDIA also follows this trend through the introduction of DP4A and DP2A dot-product instructions into its Pascal architecture GPU. Such specialized hardware units require effort to develop efficient techniques in order to exploit its full performance. Surprisingly, this special hardware (dot-product) is not used in accelerating the cryptographic algorithms in any previous works. We found that the DP2A instruction is particularly useful in computing polynomial convolution and matrix-multiplication found in FrodoKEM and Saber KEM, which are both NIST PQC Round-3 candidates. However, utilizing dot-product instructions to compute these operations in parallel is not straightforward. This is because dot-product allows two polynomial coefficients to be packed into a single 32-bit register, wherein each thread is accessing a different portion (upper or lower 16-bit) of this 32-bit register. A naïve implementation could lead to a serious overhead in loading/storing intermediate results.

1.2 Contributions

This paper provides the first implementation that utilizes the dot-product instruction in modern 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 rely only on conventional 32-bit integer units. Contributions of this paper are summarized below:

1. We propose a highly optimized matrix-multiplication implementation on GPU in which the DP2A instruction is used to speed-up dot-product operations between two matrices. The proposed data packing technique allows the polynomial coefficients to be loaded from and stored to the global memory efficiently. Combining this packing technique with the dot-product instructions, our implementation is able to accelerate the matrix-multiplication up-to 1.37×, 1.83×, and 1.58× compared to 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 is applied to FrodoKEM (i.e., DPFrodo), achieving a 4.37× speed-up compared to the state-of-the-art implementation of FrodoKEM [GJCC21] on V100 GPU platform.

2. Polynomial convolution in Saber exhibits similar computational pattern like the matrix-multiplication. Unlike ordinary polynomial multiplication, the polynomial convolution requires coefficients to be read in a cyclic form, making it non-trivial to utilize the dot-product instruction. To alleviate this limitation, we propose a novel data structure that allows the polynomial data to be read in cyclic form, at the same time achieving 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 (i.e., DPSaber) can achieve up-to 1.63×, 1.28×, and 1.54× 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 is applied to the standard Saber with parameter set $N = 256$ and $l = 3$, where the achieved key exchange throughputs are 124,418, 120,463, and 20,225 on RTX3080, V100, and T4 GPUs, respectively. This is also the first implementation of Saber on GPU platforms.

In summary, the proposed 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 $(2,048 <$

Table 1: Overview of FrodoKEM [ABD+20] and Saber [DASS+20] parameters.

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

$q \leq 65,536$), which cannot take advantage of the tensor-core-based solution proposed by Lee et al. [LSZH21]. Proposed DPFrodo and DPSaber implementations support both ephemeral and non-ephemeral key usage, which is more flexible compared to Lee et al. [LSZH21]. They support high throughput KEM, which is beneficial to conventional client-server based Internet communication, as well as the emerging Internet of Things (IoT) applications. Implementation codes discussed in this paper can be found in https://anonymous.4open.science/r/DPCrypto-D242/.

2 Background

In this section, we provide an introduction to selected lattice-based cryptographic schemes (i.e., FrodoKEM and Saber) and the related hard problems. Following this, we also present a summary of the key features in modern NVIDIA GPU architectures.

2.1 FrodoKEM

FrodoKEM is a lattice-based KEM that relies on the hardness of learning with errors (LWE) problems. It was firstly introduced as a key exchange protocol in [BCD+16] and later on developed into a KEM. FrodoKEM was selected as an alternate candidate in Round-3 of NIST PQC standardization. The main performance bottleneck in FrodoKEM comes from the matrix multiplication.

2.2 Saber

Saber 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 [BPR12] rather than the learning with errors [Reg04]. The advantage of the former over the later is that the error term is generated inherently in the former case whereas it needs to be added in the latter case. This results in lesser requirements of pseudorandom 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 [KRSS, BMK+21, MKV20, DKRV18] of Saber.

2.3 Overview of NVIDIA GPU Architecture

2.3.1 CUDA Programming Model

A GPU hardware has multiple *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 SDK released by NVIDIA to ease programming of the 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 the 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 cause a serious performance penalty.

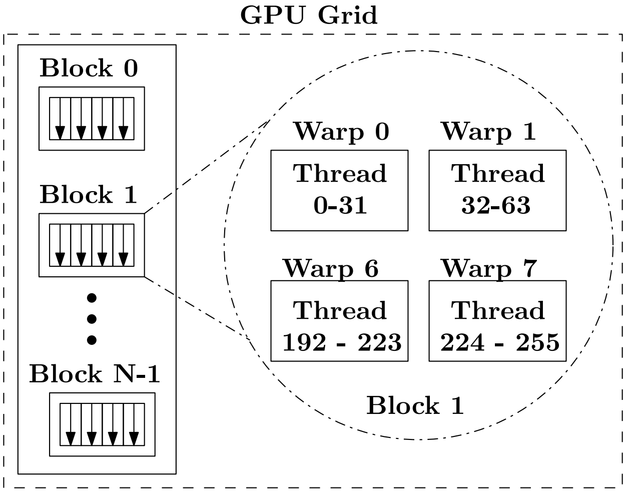


Figure 1: Relationship between grid, block, warp, and thread in CUDA.

2.3.2 Memory Hierarchy

Similar to many CPU architectures, GPU also has a deep memory hierarchy. There are two types of GPU memory in general, which have 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 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 comes with a very limited size (e.g., 64K words per SM for the RTX3080).

2.3.3 Dot-Product Instructions

Dot-product instructions were firstly introduced into Pascal architecture (i.e., compute capability 6.1), and they are supported in the subsequent GPU architectures. Referring to Figure 2, there are two versions of dot-product instructions in NVIDIA GPU. 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. DP2A comes with two variants; they are either operated on the first (DP2A_hi) or the last (DP2A_lo) two 8-bit inputs. Both versions support signed and unsigned operands.

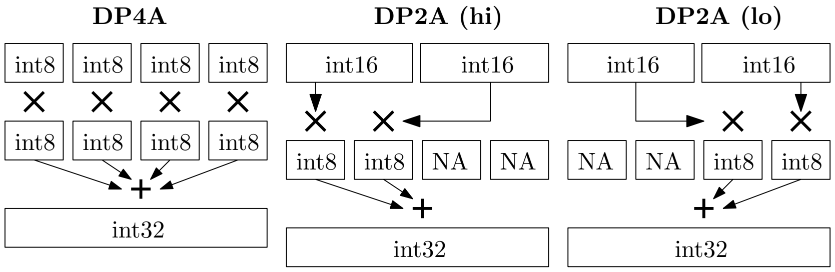


Figure 2: Dot-product instructions in NVIDIA GPU.

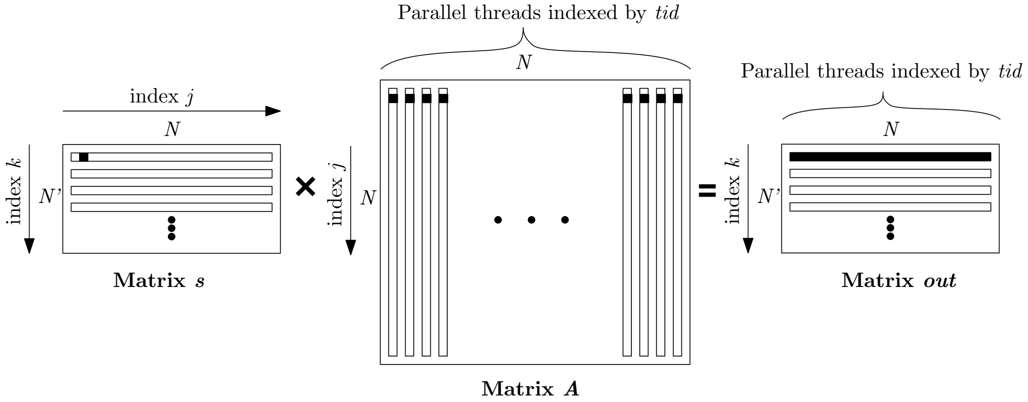


Figure 3: Parallel implementation of matrix multiplication in FrodoKEM.

2.4 Related Work

The first FrodoKEM implementation on GPU was presented by Gupta et al. [GJCC21]. Authors utilized *single* mode to compute FrodoKEM, wherein multiple blocks and multiple threads cooperatively execute algorithms on 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 the high amount of atomic instructions. Moreover, FrodoKEM can be computed by using 16-bit coefficients, but Gupta et al. [GJCC21] only utilize the 32-bit integer units, which is not an optimal choice.

Another two notable works published recently are from Lee et al. [LH21] and Gao et al. [GXW21]. These works showcased high throughput implementations of Kyber and NewHope KEM on GPU platforms, which rely on the use NTT. However, these works also do not use advanced features (e.g., dot-product instructions) found in modern GPU architectures.

In this paper, we show that FrodoKEM can achieve a higher throughput through a more optimized matrix-matrix multiplication technique and the use of dot-product instructions. We also showcase the first optimized implementation of Saber on various GPU platforms.

3 GPU Implementation Techniques

In this section, we present details of GPU implementation techniques targeting two selected lattice-based cryptographic schemes.

3.1 Highly Optimized (INT32) and Dot-product Aided (DPFrodo) Matrix Multiplication Implementations of FrodoKEM on GPU

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. 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 matrix A are multiplied with a value from matrix s, and then added to results in matrix *out*. The 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 5 and 7 in GPU. However, this naïve implementation causes a lot of read/write into the global memory, seriously limiting the performance of implementation in GPU.

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                      ▷ $N \times N'$ is the size of matrix
4:         $load\_a$ = A[$j \times N + tid$];              ▷ $tid$ is the thread ID
5:         for ($k = 0; k < N'; k++$) do
6:             out[$k \times N + tid$] += $load\_a \times$ s[$k \times N + j$];
7:         end for
8:     end for
9: end procedure
```

Algorithm 2 Unrolled parallel matrix multiplication in FrodoKEM.

```
procedure MAT_MUL_UNROLL($out, A, s$)
    $sum0 = 0$, $sum1 = 0$, $sum2 = 0$, $sum3 = 0$;
    $sum4 = 0$, $sum5 = 0$, $sum6 = 0$, $sum7 = 0$;       ▷ $N \times N'$ is the size of matrix
    for ($j = 0; j < N; j++$) do
        $load\_a$ = A[$j * N + tid$];                      ▷ $tid$ is the thread ID
        $sum0$ += $load\_a \times$ s[$j$];                 ▷ Unroll 8 times ($N'$)
        $sum1$ += $load\_a \times$ s[$1 \times N + j$];
        $sum2$ += $load\_a \times$ s[$2 \times N + j$];
        $sum3$ += $load\_a \times$ s[$3 \times N + j$];
        ...                                                ▷ Removed for brevity
    end for
    out[$tid$] = $sum0$;                                   ▷ Unroll 8 times ($N'$)
    out[$1 \times N + tid$] = $sum1$;
    out[$2 \times N + tid$] = $sum2$;
    out[$3 \times N + tid$] = $sum3$;
    ...                                                    ▷ Removed for brevity
end procedure
```

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 [ABD+20]. Hence, it is possible to fully unroll the k loop in Algorithm 1. By doing this, we can exploit the use of more registers during the computation of multiply-and-add operations. Algorithm 2 shows this improved implementation technique.

Referring to Table 1, the modulus q is either 32,768 or 65,536. This implies that the

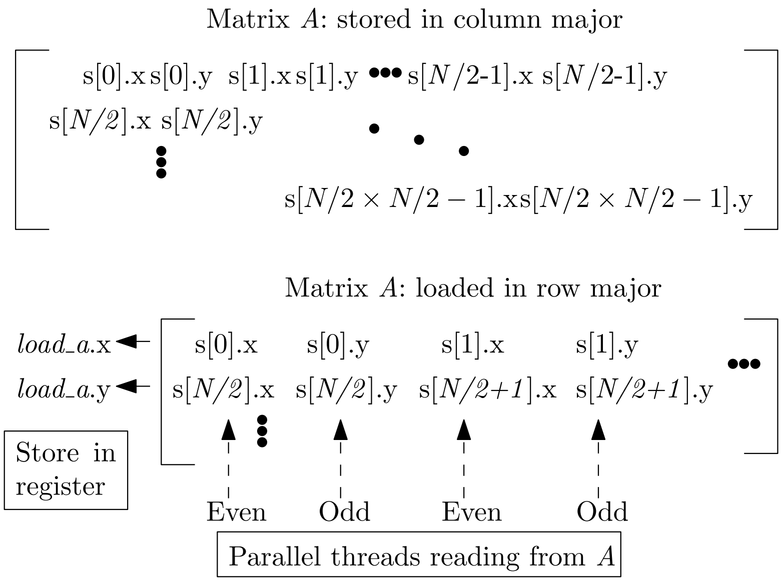

Matrix A: stored in column major
s[0].x s[0].y s[1].x s[1].y ●●● s[N/2-1].x s[N/2-1].y
s[$N/2$].x s[$N/2$].y
s[$N/2 \times N/2 - 1$].x s[$N/2 \times N/2 - 1$].y
Matrix A: loaded in row major
$load\_a$.x
$load\_a$.y
s[0].x s[0].y s[1].x s[1].y
s[$N/2$].x s[$N/2$].y s[$N/2+1$].x s[$N/2+1$].y
Store in register
Even
Odd
Even
Odd
Parallel threads reading from A

Figure 4: Parallel implementation of matrix multiplication in FrodoKEM.

matrix multiplication and accumulation can be carried out entirely on a 16-bit variable without causing any error. In this paper, we exploited this property and proposed a more optimized implementation to push the performance of FrodoKEM matrix multiplication to the limit. Note that the Matrix A was originally stored in a column major and read in a row major; 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 3 DPFrodo: packing the matrix s.

```
procedure PACK_MAT_S($s_{packed}$, $s$)
    for ($i = 0; i < N'; i++$) do                              ▷ N' is 8 for FrodoKEM
        $s_{packed}[i \times N/2 + tid].x = s[i \times N + 2 \times tid]$;    ▷ tid is the thread ID
        $s_{packed}[i \times N/2 + tid].y = s[i \times N + 2 \times tid + 1]$;
    end for
end procedure
```

The proposed parallel implementation of matrix-matrix multiplication in FrodoKEM using the dot-product instruction is detailed in Algorithm 4. The j (lines 4 ∼ 20) loop is executed to accumulate results of matrix multiplication. In each iteration, even threads load x components of matrix A (lines 5 ∼ 7), while odd threads load y components (lines 8 ∼ 10). This is followed by multiplications between matrix A and b (lines 12 ∼ 19), which is fully unrolled by $N' = 8\times$. Finally, results are stored in the output array (lines 21 ∼ 25), which marks the end of the Algorithm 4.

3.2 The First Optimized (INT32) and Dot-product Aided (DPSaber) Polynomial Convolution Implementations of Saber on GPU

The most time-consuming operation in Saber is the polynomial convolution, which is detailed in Algorithm 5. The j loop (lines 2 ∼ 10) iterates through all coefficients in the

Algorithm 4 DPFrodo: parallel matrix multiplication in FrodoKEM with dot-product instruction.

```
procedure Mat_Mul_and_Add_Unroll($out, A, s$)
    $sum0 = 0$, $sum1 = 0$, $sum2 = 0$, $sum3 = 0$;
    $sum4 = 0$, $sum5 = 0$, $sum6 = 0$, $sum7 = 0$;
    for ($j = 0; j < N/2; j++$) do                    ▷ $N \times N'$ is the size of matrix
        if $tid\%2 == 0$ then                         ▷ $tid$ is the thread ID
            $load\_a.x = A[j * N + tid/2].x$;
            $load\_a.y = A[j * N + N/2 + tid/2].x$;
        else
            $load\_a.x = A[j * N + tid/2].y$;
            $load\_a.y = A[j * N + N/2 + tid/2].y$;
        end if
        $sum0 \mathrel{+}= load\_a.x \times s[j].x + load\_a.y \times s[j].y$;
        $sum1 \mathrel{+}= load\_a.x \times s[1 \times N + j].x +$
            $load\_a.y \times s[1 \times N + j].y$;
        $sum2 \mathrel{+}= load\_a.x \times s[2 \times N + j].x +$
            $load\_a.y \times s[2 \times N + j].y$;
        $sum3 \mathrel{+}= load\_a.x \times s[3 \times N + j].x +$
            $load\_a.y \times s[3 \times N + j].y$;
        ...                                           ▷ Unroll 8 times ($N$'). Removed for brevity
    end for
    out[$tid$] = $sum0$;
    out[$1 \times N + tid$] = $sum1$;
    out[$2 \times N + tid$] = $sum2$;
    out[$3 \times N + tid$] = $sum3$;
    ...                                               ▷ Unroll 8 times ($N$'). Removed for brevity
end procedure
```

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 degree of polynomial
 3:         $sum = 0$;
 4:         for ($i = 0; i < j{+}1; i{+}{+}$) do          ▷ Accumulation
 5:             $sum = sum + a[j - i] \times b[i]$;
 6:         end for
 7:         for ($i = 1; i < N{-}j; i{+}{+}$) do          ▷ Subtraction
 8:             $sum = sum - a[j + i] \times b[N - i]$;
 9:         end for
10:         $out[j] = sum$;                               ▷ $out$ is the array to store the final results
11:     end for
12: end procedure
```

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 GPU, where the polynomial degree $N = 8$. Since each thread can perform the multiplication and accumulation independently, this technique is considered quite efficient for the GPU implementation. Recently, Lee et al. [LSZH21] proposed an improved version of this technique, wherein multiplication and accumulation operations are represented in a matrix form and computed entirely on tensor cores. The tensor-core-based solution is able to

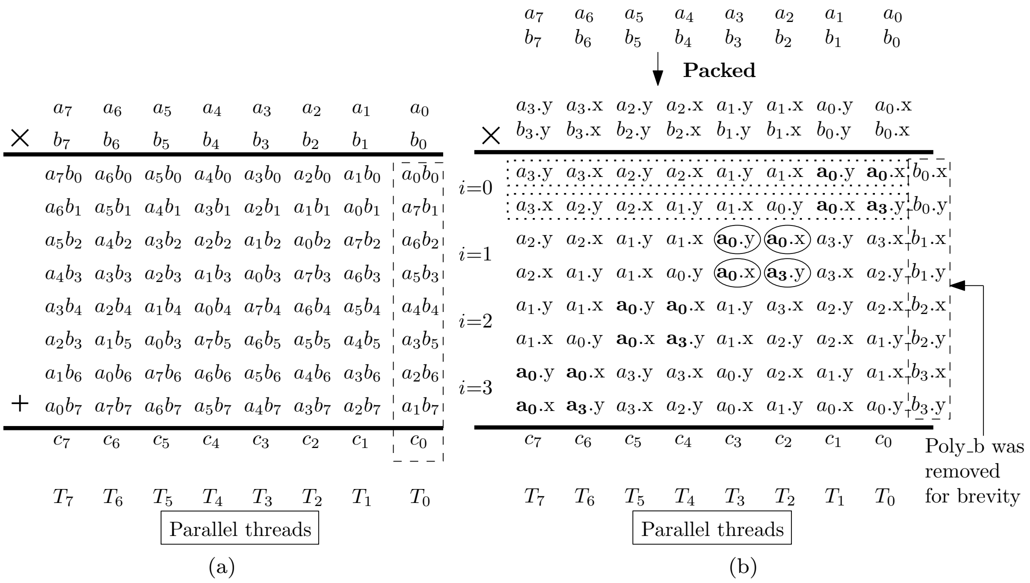

a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
× b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
a_7b_0 a_6b_0 a_5b_0 a_4b_0 a_3b_0 a_2b_0 a_1b_0 a_0b_0
a_6b_1 a_5b_1 a_4b_1 a_3b_1 a_2b_1 a_1b_1 a_0b_1 a_7b_1
a_5b_2 a_4b_2 a_3b_2 a_2b_2 a_1b_2 a_0b_2 a_7b_2 a_6b_2
a_4b_3 a_3b_3 a_2b_3 a_1b_3 a_0b_3 a_7b_3 a_6b_3 a_5b_3
a_3b_4 a_2b_4 a_1b_4 a_0b_4 a_7b_4 a_6b_4 a_5b_4 a_4b_4
a_2b_3 a_1b_5 a_0b_3 a_7b_5 a_6b_5 a_5b_5 a_4b_5 a_3b_5
a_1b_6 a_0b_6 a_7b_6 a_6b_6 a_5b_6 a_4b_6 a_3b_6 a_2b_6
+ a_0b_7 a_7b_7 a_6b_7 a_5b_7 a_4b_7 a_3b_7 a_2b_7 a_1b_7
c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
T_7 T_6 T_5 T_4 T_3 T_2 T_1 T_0
Parallel threads
(a)
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Packed
a_3.y a_3.x a_2.y a_2.x a_1.y a_1.x a_0.y a_0.x
× b_3.y b_3.x b_2.y b_2.x b_1.y b_1.x b_0.y b_0.x
i=0
a_3.y a_3.x a_2.y a_2.x a_1.y a_1.x $\mathbf{a_0}$.y $\mathbf{a_0}$.x b_0.x
a_3.x a_2.y a_2.x a_1.y a_1.x a_0.y $\mathbf{a_0}$.x $\mathbf{a_3}$.y b_0.y
i=1
a_2.y a_2.x a_1.y a_1.x $\mathbf{a_0}$.y $\mathbf{a_0}$.x a_3.y a_3.x b_1.x
a_2.x a_1.y a_1.x a_0.y $\mathbf{a_0}$.x $\mathbf{a_3}$.y a_3.x a_2.y b_1.y
i=2
a_1.y a_1.x $\mathbf{a_0}$.y $\mathbf{a_0}$.x a_1.y a_3.x a_2.y a_2.x b_2.x
a_1.x a_0.y $\mathbf{a_0}$.x $\mathbf{a_3}$.y a_1.x a_2.y a_2.x a_1.y b_2.y
i=3
$\mathbf{a_0}$.y $\mathbf{a_0}$.x a_3.y a_3.x a_0.y a_2.x a_1.y a_1.x b_3.x
$\mathbf{a_0}$.x $\mathbf{a_3}$.y a_3.x a_2.y a_0.x a_1.y a_0.x a_0.y b_3.y
c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
Poly_b was removed for brevity
T_7 T_6 T_5 T_4 T_3 T_2 T_1 T_0
Parallel threads
(b)

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

achieve the polynomial convolution with high throughput, but it can only support the polynomial convolution for lattice-based schemes that utilize a small modulus q. The modulus q must be lesser than 2^{11}, due to the limitation in half precision floating point arithmetic. Hence, it is not suitable to be used in Saber.

Referring to Table 1, the modulus q of Saber parameter sets is always $8,192 = 2^{13}$. Considering the Saber parameter set, coefficients of small polynomial (b) are in the range of -4 to +4, which can be conveniently represented in an 8-bit variable. Coefficients of polynomial (a), which 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 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 reads the x component from 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 threads. Odd threads (i.e., $T_1, T_3, ...$) also need to access a different component from even threads. These access patterns cause many conditional statements (e.g., if/else) in a naïve GPU implementation, since each thread needs to decide whether to read the x or y component, as well as deciding the right index to read the polynomial a. Note that this problem is not found in polynomial b, because it is always accessed in a sequential and fixed manner.

In this paper, we proposed a novel data structure to hold the polynomial a in order to avoid the problems mentioned above. A closer look into Figure 5b reveals that all the even threads are reading different values, but they exhibit a cyclic pattern. For instance, when $i = 0$, T_0 reads a_0.x and a_3.y; the same pair of values are being read by T_2, T_4, and

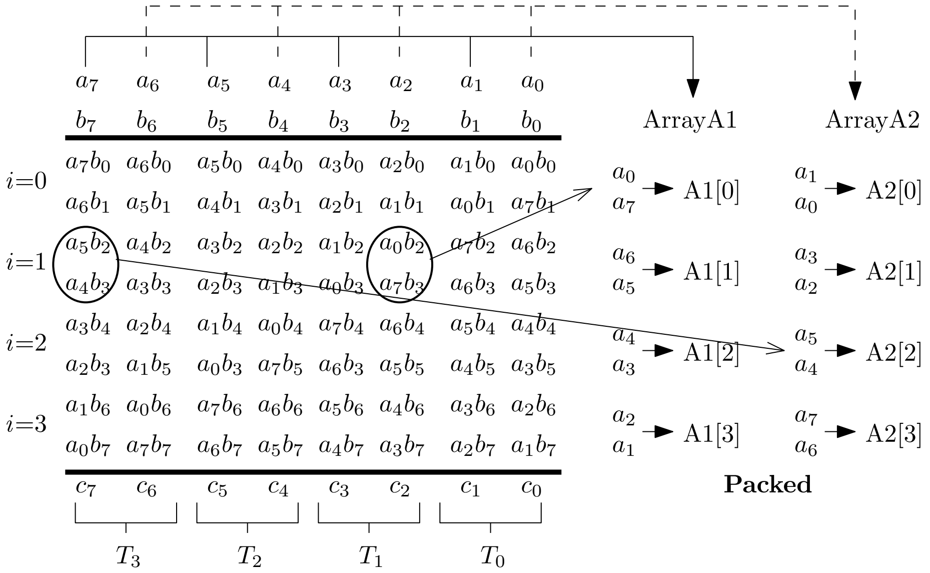
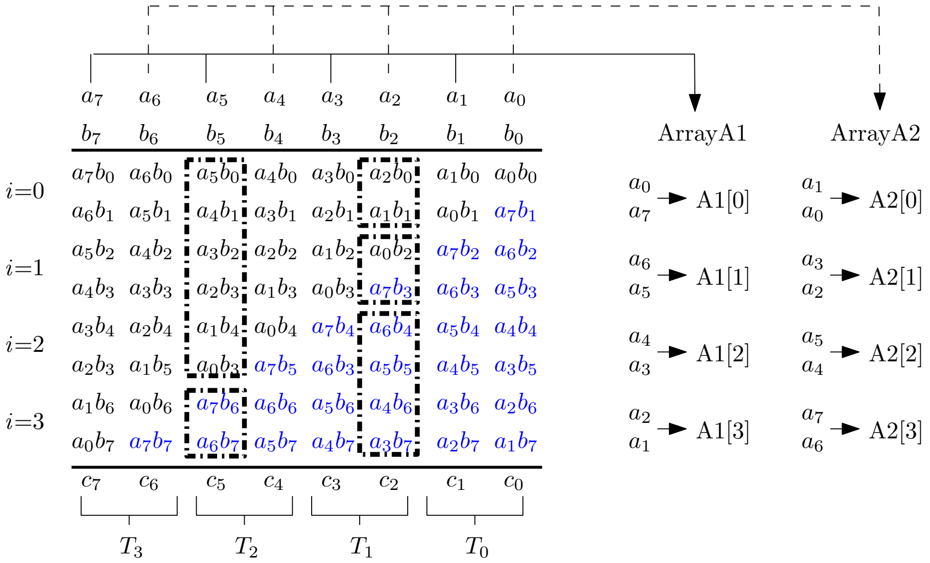

a7 a6 a5 a4 a3 a2 a1 a0
b7 b6 b5 b4 b3 b2 b1 b0
ArrayA1
ArrayA2
i=0
a7b0 a6b0 a5b0 a4b0 a3b0 a2b0 a1b0 a0b0
a6b1 a5b1 a4b1 a3b1 a2b1 a1b1 a0b1 a7b1
i=1
a5b2 a4b2 a3b2 a2b2 a1b2 a0b2 a7b2 a6b2
a4b3 a3b3 a2b3 a1b3 a0b3 a7b3 a6b3 a5b3
i=2
a3b4 a2b4 a1b4 a0b4 a7b4 a6b4 a5b4 a4b4
a2b3 a1b5 a0b3 a7b5 a6b3 a5b5 a4b5 a3b5
i=3
a1b6 a0b6 a7b6 a6b6 a5b6 a4b6 a3b6 a2b6
a0b7 a7b7 a6b7 a5b7 a4b7 a3b7 a2b7 a1b7
c7 c6 c5 c4 c3 c2 c1 c0
T3 T2 T1 T0
a0 a7 A1[0]
a6 a5 A1[1]
a4 a3 A1[2]
a2 a1 A1[3]
a1 a0 A2[0]
a3 a2 A2[1]
a5 a4 A2[2]
a7 a6 A2[3]
Packed

Figure 6: Proposed packing method for polynomial convolution in Saber.

T_6 at different time (See bolded parts). The same patterns also apply to the odd 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. To utilize this proposed data structure, we also need half the number of parallel threads (from $N = 256$ to $N = 128$, where N is the degree of polynomial in Saber).

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, depending on the thread index and index i. For instance, considering the case of odd threads (i.e., $T_1, T_3, ...$), 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 threads, wherein there are 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. This also implies that the implementation of parallel polynomial convolution in Saber is more complicated compared to FrodoKEM, as there are more conditional checks required.

Detailed implementation steps are presented in Algorithm 6. Firstly, packed polynomials are loaded into shared memory to improve the accessing speed (lines 3 ∼ 5). Following this, the algorithm loads a value from *s_A2* and proceeds to compute the dot-product operation (lines 8 ∼ 15). To process even threads, there are three different conditions, which should be checked (lines 8, 10, and 13). This corresponds to the situations explained in Figure 7 for even threads. The computation for odd 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



a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
ArrayA1
ArrayA2
i=0
a_7b_0 a_6b_0 a_5b_0 a_4b_0 a_3b_0 a_2b_0 a_1b_0 a_0b_0
a_6b_1 a_5b_1 a_4b_1 a_3b_1 a_2b_1 a_1b_1 a_0b_1 a_7b_1
i=1
a_5b_2 a_4b_2 a_3b_2 a_2b_2 a_1b_2 a_0b_2 a_7b_2 a_6b_2
a_4b_3 a_3b_3 a_2b_3 a_1b_3 a_0b_3 a_7b_3 a_6b_3 a_5b_3
i=2
a_3b_4 a_2b_4 a_1b_4 a_0b_4 a_7b_4 a_6b_4 a_5b_4 a_4b_4
a_2b_5 a_1b_5 a_0b_5 a_7b_5 a_6b_5 a_5b_5 a_4b_5 a_3b_5
i=3
a_1b_6 a_0b_6 a_7b_6 a_6b_6 a_5b_6 a_4b_6 a_3b_6 a_2b_6
a_0b_7 a_7b_7 a_6b_7 a_5b_7 a_4b_7 a_3b_7 a_2b_7 a_1b_7
c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
T_3 T_2 T_1 T_0
a_0 a_7 A1[0]
a_6 a_5 A1[1]
a_4 a_3 A1[2]
a_2 a_1 A1[3]
a_1 a_0 A2[0]
a_3 a_2 A2[1]
a_5 a_4 A2[2]
a_7 a_6 A2[3]

Figure 7: Computing nega-cyclic polynomial convolution in Saber.

Algorithm 6 DPSaber: parallel polynomial convolution in Saber with dot-product instruction.

```
procedure POLY_CONV($out, A1, A2, b$)
    $sum1 = 0$, $sum2 = 0$;
    __shared__ $s\_A1[tid] = A1[tid]$;                          ▷ $tid$ is the thread ID
    __shared__ $s\_A2[tid] = A2[tid]$;
    __shared__ $s\_b[tid] = b[tid]$;
    for $(i = 0; i < N/2; i++)$ do                 ▷ $N$ is the degree of polynomial
        $load\_a = s\_A2[(tid \times (N/2 - 1) + i)\%(N/2)]$;
        ▷ Processing the even elements.
        if $i > tid$ then
            $sum1$ -= $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
        else if $i == tid$ then
            $sum1$ += $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
            $sum1$ -= $load\_a.y \times s\_b[i].y + load\_a.y \times s\_b[i].y$;
        else
            $sum1$ += $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
        end if
        $load\_a = s\_A1[(tid + i \times (N/2 - 1))\%N/2]$;
        ▷ Processing the odd elements.
        if $i \leq tid$ then
            $sum2$ += $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
        else
            $sum2$ -= $load\_a.x \times s\_b[i].x + load\_a.y \times s\_b[i].y$;
        end if
    end for
    out$[tid \times 2] = sum1$;
    out$[tid \times 2 + 1] = sum2$;
end procedure
```

Table 2: Experimental platforms for proposed implementations.

Platforms	CPU	Clock (GHz)	RAM (GB)	GPU	Clock (GHz)	Mem. BW (GB/s)
A	i9-10900K	3.7	16	RTX3080	1.44	760.3
B	Xeon Gold 5120	2.2	16	V100	1.25	897.0
				T4	0.585	320.0

positions (lines 23 ∼ 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.

4 Experimental Results

This section presents the 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 2. 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) [com21], 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. Note that the three selected GPU (V100, T4 and RTX3080) represents the state-of-the-art GPU architectures (Volta, Turing and Ampere) from NVIDIA that supports dot-product instructions. The AVX2 implementation was executed on the i9-10900K CPU, configured to clock at 3.7 GHz.

In this paper, we have selected the FrodoKEM976 and Saber parameter sets belonging to the NIST security category 3 for the performance evaluation. All implementations follow the fine-grain parallel approach, wherein one block consists of multiple threads that are used to complete one KEM or one matrix/vector operation. On top of that, many parallel blocks are initiated, where the number of parallel blocks, K, varies. In each experiment, K increases gradually to observe the achieved throughput, until the performance saturates.

4.1 Evaluation of Proposed FrodoKEM Implementations (INT32 and DPFrodo)

Table 3: Comparison of proposed matrix-matrix multiplication implementations in FrodoKEM976 based on int 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	29270	1.98
128	173319	215922	1.25	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 3 shows the throughput of computing one matrix multiplication in FrodoKEM using different implementation techniques. Results show that DPFrodo is at least 1.25× faster than the conventional implementation using integer unit (INT32), across different K on various GPU platforms. Performance saturates when K is relatively large (≥ 512), indicating that further increasing the number of parallel blocks does not help in improving performance anymore.

Table 4: Comparison of proposed FrodoKEM976 scheme implementations based on int 32-bit integer units (INT32) and DP2A (DPFrodo) instructions.

RTX 3080								
K	Encap Throughput			Decap Throughput			KX/s	
	INT32	DPFrodo	Sp-Up	INT32	DPFrodo	Sp-Up	INT32	DPFrodo
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**
[GJCC21]		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		1428			1495		731	

The proposed DPFrodo technique is applied to FrodoKEM parameter set FrodoKEM976 to speed up the matrix multiplication. Referring to Table 4, DPFrodo is able to produce 1.07× higher throughput for both RTX 3080 and V100, and 1.05× for T4, against the conventional implementation utilizing 32-bit integer units. Compared to the state-of-the-art results produced by the implementation from Gupta et al. [GJCC21], our proposed implementation is able to achieve 4.37× higher throughput in terms of key exchanges per second on the same GPU (V100). The main reason for this performance gain is due to the proposed data packing technique that allows the dot-product computation to be carried out efficiently, which is better than using conventional 32-bit integer units. Besides that, their implementation uses many blocks and threads to compute one FrodoKEM, which requires many communication between threads. This also requires the ues of atomic instructions to avoid data hazard, which is a huge overhead. Our implemenation avoid this problem by computing many FrodoKEMs with multiple blocks, and each block computes one FrodoKEM. The communication between threads is reduced, at the same time avoided the use of atomic instructions. The DPFrodo key exchange throughput is also higher than the AVX2 implementation by 8.49×, 6.68×, and 3.61×, on RTX3080, V100, and T4 GPU platforms, respectively.

4.2 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 5 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

Table 5: 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

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 $K \geq 1024$. Overall, the speed-up gained 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 the polynomials, which introduced some overhead. Hence, the memory to compute ratio has to be large enough in order to capitalize the 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 6 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, 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 is 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 throughput achieved by DPSaber implementation is 4.17×, 4.04×, and 1.06× higher than AVX2 implementation on RTX 3080, V100, and T4, respectively.

4.3 Practical Use Cases of Proposed GPU Implementation

4.3.1 Secure Cloud Computing

Cloud computing is the fundamental technology that supports many important online activities involving various facets in our daily life. Some online activities that involve sensitive data need to be protected by security protocols. Key exchange is a fundamental feature supported by many security protocols such as SSL/TLS [RD18] and IPsec (IKE) [KHN+10]. 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

Table 6: Comparison of proposed Saber KEM implementations based on int 32-bit integer units (INT32) and DP2A (DPSaber) instructions.

	RTX 3080							
K	Encap Throughput			Decap Throughput			KX/s	
	INT32	DPSaber	Sp-Up	INT32	DPSaber	Sp-Up	INT32	DPSaber
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	52835			53981			29836	

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 our proposed solution, we can effectively offload batch computations of KEMs to GPU and eventually save a lot of time in CPU, which thus allows the server to execute other tasks. Note that GPUs are already commonly found in many major cloud computing services like AWS [AWS21] and IBM [IBM21], 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, which are more rarely available or costlier.

4.3.2 Secure Internet of Things Communication

Another interesting use case is the IoT, which is an emerging and paradigm shifting technology. In many IoT applications, the sensor nodes are actively interacting with the cloud servers. The scale of IoT system ranges from hundreds to thousands of sensor nodes [CGD+20]. To secure such a communication, symmetric keys used to encrypt the sensor data, need to be refreshed frequently, which can be done through one of the following methods:

1. New session keys are produced by the IoT sensor nodes and transmitted to the cloud server via KEM. Typically, the symmetric key is refreshed in every communication session using pseudo random number generator (PRNG) 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.

Method 1 requires the cloud server to decapsulate and obtain session keys, while method 2 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 other computations. For instance, this was previously discussed in a recent publication by Lee et al. [LH21], wherein the concept of KEM-as-a-service (KEDaaS) was proposed to reduce the burden of the cloud server. The success of such proposal relies heavily on the high throughput performance of KEM implemented on GPU. We believe that our proposed implementation techniques are very suitable to this use case.

5 Conclusion

In this paper, we 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 was presented, which allows the proposed FrodoKEM implementation to be 4.37× faster than the state-of-the-art work proposed by Gupta et al. [GJCC21]. A novel data structure was also proposed to allow the parallel polynomial convolution to be computed efficiently using the DP2A instruction. Note that these two proposed techniques are generic; 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) [rdn], 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 [LLZ+18], which is a NIST round 2 candidate.

An interesting extension from this work is to explore possibilities to use dot-product instruction for NTT computation, which is widely used for some NIST finalist candidates (e.g., Kyber and Dillithium). A more optimized implementation of random samples generation (through AES or SHAKE) can also help in improving the performance of lattice-based cryptographic schemes on GPU platforms.

References

[ABD+20] Erdem Alkim, Joppe W Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM learning with errors key encapsulation, 2020.

[ABPA+19] Ahmad Qaisar Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. *IEEE Transactions on Emerging Topics in Computing*, 2019.

[ABVL+20] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):379–391, 2020.

[ABVMA18] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 70–95, 2018.

[ARM] A64 – SVE instructions.

[ASL+20] Sedat Akleylek, Meryem Soysaldı, Wai-Kong Lee, Seong Oun Hwang, and Denis Chee-Keong Wong. Novel post-quantum MQ-based signature scheme for Internet of Things with parallel implementation. *IEEE Internet of Things Journal*, 2020.

[AWS21] Amazon ec2 p4d instances. https://aws.amazon.com/ec2/instance-types/p4/, 2021. Accessed: 2021-10-10.

[BC18] Avishek Biswas and Anantha P Chandrakasan. CONV-SRAM: An energy-efficient SRAM with in-memory dot-product computation for low-power convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 54(1):217–230, 2018.

[BCD+16] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018, 2016.

[BMK+21] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhede. Polynomial multiplication on embedded vector architectures. Cryptology ePrint Archive, Report 2021/998, 2021. https://ia.cr/2021/998.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *EUROCRYPT 2012*, pages 719–737, 2012.

[CGD+20] Flavio Cirillo, David Gómez, Luis Diez, Ignacio Elicegui Maestro, Thomas Barrie Juel Gilbert, and Reza Akhavan. Smart city IoT services creation through large-scale collaboration. *IEEE Internet of Things Journal*, 7(6):5267–5275, 2020.

[com21] Compute Canada. https://www.computecanada.ca/home/, 2021. Accessed: 2021-10-10.

[DASS+20] Jan-Pieter D'Anvers, Karmakar Angshuman, Roy Sujoy Sinha, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. Saber: Mod-LWR based kem, 2020.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*, volume 10831 of *Lecture Notes in Computer Science*, pages 282–305. Springer, 2018.

[DSA+21] Shi Dong, Yifan Sun, Nicolas Bohm Agostini, Elmira Karimi, Daniel Lowell, Jing Zhou, José Cano, José L Abellán, and David Kaeli. Spartan: A sparsity-adaptive framework to accelerate deep neural network training on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2448–2463, 2021.

[GJCC21] Naina Gupta, Arpan Jati, Amit Kumar Chauhan, and Anupam Chattopadhyay. PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):575–586, 2021.

[GXW21] Yiwen Gao, Jia Xu, and Hongbing Wang. cuNH: Efficient GPU implementations of post-quantum KEM NewHope. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):551–568, 2021.

[HMKG19] Omid Hajihassani, Saleh Khalaj Monfared, Seyed Hossein Khasteh, and Saeid Gorgin. Fast AES implementation: A high-throughput bitsliced approach. *IEEE Transactions on parallel and distributed systems*, 30(10):2211–2222, 2019.

[IBM21] NVIDIA GPUs on IBM cloud servers. https://www.ibm.com/cloud/gpu, 2021. Accessed: 2021-10-10.

[KHN$^+$10] Charlie Kaufman, Paul Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet key exchange protocol version 2 (ikev2). Technical report, RFC 5996, September, 2010.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[LGP19] Wai-Kong Lee, Bok-Min Goi, and Raphael C-W Phan. Terabit encryption in a second: Performance evaluation of block ciphers in GPU with Kepler, Maxwell, and Pascal architectures. *Concurrency and Computation: Practice and Experience*, 31(11):e5048, 2019.

[LGZ$^+$19] Xinya Lei, Ruixin Guo, Feng Zhang, Lizhe Wang, Rui Xu, and Guangzhi Qu. Optimizing FHEW with heterogeneous high-performance computing. *IEEE Transactions on Industrial Informatics*, 16(8):5335–5344, 2019.

[LH21] Wai Kong Lee and Seong Oun Hwang. High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of Things applications. *IEEE Transactions on Services Computing*, 2021.

[LLZ$^+$18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, Kunpeng Wang, Zhe Liu, and Hao Yang. LAC: Practical Ring-LWE based public-key encryption with byte-level modulus. *IACR Cryptol. ePrint Arch.*, 2018:1009, 2018.

[LSZH21] Wai-Kong Lee, Hwa-Jeong Seo, Zhenfei Zhang, and Seongoun Hwang. Tensor Crypto. *IACR Eprint*, https://eprint.iacr.org/2021/173, 2021.

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020, Issue 2:222–244, 2020.

[NCB$^+$20] Cristóbal A Navarro, Roberto Carrasco, Ricardo J Barrientos, Javier A Riquelme, and Raimundo Vega. GPU tensor cores for fast arithmetic reductions. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):72–84, 2020.

[NIS] Post-quantum cryptography: Round 3 submissions.

[NIS17] NIST. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization, 2017. [Online; accessed 10-Oct-2021].

[RD18] Eric Rescorla and Tim Dierks. The transport layer security (tls) protocol version 1.3. 2018.

[rdn] RDNA 2" instruction set architecture, reference guide, advanced micro devices, inc. https://developer.amd.com/wp-content/resources/RDNA2_Shader_ISA_November2020.pdf. Accessed: 2021-09-30.

[Reg04] Oded Regev. *New Lattice-based Cryptographic Constructions*, volume 51-6, pages 899–942. ACM, New York, NY, USA, November 2004.

[SHLW19] Xueneng Su, Chuan He, Tianqi Liu, and Lei Wu. Full parallel power flow solution: A GPU-CPU-based vectorization parallelization and sparse techniques for newton–raphson implementation. *IEEE Transactions on Smart Grid*, 11(3):1833–1844, 2019.

[SZM20] Shuzhou Sun, Rui Zhang, and Hui Ma. Efficient parallelism of post-quantum signature scheme sphincs. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2542–2555, 2020.

[TBY+20] Nazanin Tahmasebi, Pierre Boulanger, Jihyun Yun, Gino Fallone, Michelle Noga, and Kumaradevan Punithakumar. Real-time lung tumor tracking using a CUDA enabled nonrigid registration algorithm for MRI. *IEEE journal of translational engineering in health and medicine*, 8:1–8, 2020.

[Tez21] Cihangir Tezcan. Optimization of advanced encryption standard on graphics processing units. *IEEE Access*, 9:67315–67326, 2021.

[VSG+20] Jure Vreča, Karl JX Sturm, Ernest Gungl, Farhad Merchant, Paolo Bientinesi, Rainer Leupers, and Zmago Brezočnik. Accelerating deep learning inference in constrained embedded devices using hardware loops and a dot product unit. *IEEE Access*, 8:165913–165926, 2020.