cuFE: High Performance Privacy Preserving Support Vector Machine with Inner-Product Functional Encryption

KyungHyun Han[1], Wai-Kong Lee[2], Angshuman Karmakar[3], Jose Maria Bermudo Mera[3] and Seong Oun Hwang[2]

[1] Department of Electronics and Computer Engineering, Hongik University, Sejong, South Korea,
co112kr@mail.hongik.ac.kr

[2] Department of Computer Engineering, Gachon University, Seongnam, South Korea,
{waikonglee,sohwang}@gachon.ac.kr

[3] imec-COSIC, KU Leuven, Leuven, Belgium,
{Jose.Bermudo,Angshuman.Karmakar}@esat.kuleuven.be

Abstract. Privacy preservation is a sensitive issue in our modern society. It is becoming increasingly important in many applications in this ever-growing and highly connected digital era. Functional encryption is a computation on encrypted data paradigm that allows users to retrieve the evaluation of a function on encrypted data without revealing the data, thus effectively protecting users' privacy. However, existing functional encryption implementations are still very time-consuming for practical deployment, especially when applied to machine learning applications that involve huge amount of data. In this paper, we present a high performance implementation of inner-product functional encryption (IPFE) based on ring-learning with errors on graphics processing units. We propose novel techniques to parallelize the Gaussian sampling, which is one of the most time-consuming operations in the IPFE scheme. We further execute a systematic investigation to select the best strategy for implementing number theoretic transform and inverse number theoretic transform for different security levels. Compared to the existing AVX2 implementation of IPFE, our implementation on a RTX 2060 GPU device can achieve 34.24×, 40.02×, 156.30× and 18.76× speed-up for `Setup`, `Encrypt`, `KeyGen` and `Decrypt` respectively. Finally, we propose a fast privacy-preserving Support Vector Machine (SVM) application to classify data securely using our GPU-accelerated IPFE scheme. Experimental results show that our implementation can classify 100 inputs with 591 support vectors in 688 ms (less than a second), which is 33.12× faster than the AVX2 version that takes 23 seconds.

Keywords: Inner-product functional encryption · Ring-learning with errors · Graphics processing units · Support vector machines · privacy-preserving

1 Introduction

Rapid research and development in the fields of the internet of things (IoT) and web technologies have helped them to penetrate almost every facets of our society. This has made our current society producer and consumer of massive amounts of data which is unprecedented in history. This massive flow of data has ushered us into a data driven society where there is a palpable link between the data and real-world activities. Advances in data analysis and computing power have enabled us to observe, reason, and act on the data in a far better way for the overall betterment of the society. Some examples are the protection and efficient usage of our natural resources, better allocation of public funds for greater reach and minimizing loss, advancing drug discovery and medical research, etc. However, like any other breakthrough in science and technology, the advances in data analysis technologies can be also used in many insidious applications. Unfortunately, it is not very uncommon to see these technologies being used for aggressive guerilla marketing, interference in general elections, beguiling vulnerable sections of the society, etc. Therefore, it is a general consensus among public lawmakers, philanthropists, and many industry leaders that ethical and responsible use of data is an absolute necessity and measures should be put into place to enforce this. Many governing bodies have introduced laws for this purpose such as the European Union's General Data Protection Regulation, South Korea's Personal Information Protection Act, etc. However, these laws are often limited by their respective jurisdiction and do not provide an universal solution. Also they often falls short to meet their objectives against rogue and powerful entities. Computation on Encrypted Data (COED) is an umbrella term that comprises of three technologies, homomorphic encryption (HE) [Gen09], multi-party computation (MPC) [CCD88, BGW88], and functional encryption (FE) [BSW11, O'N10]. These techniques allow a user compute on encrypted data. Thus these technologies offer a universal and provable solution for maintaining privacy and integrity of their data while allowing essential and intended operations to run on them. Among these three technologies, FE is the newest and relatively less studied. Unlike the other two COED techniques which can theoretically compute arbitrary functions, FE schemes are designed for some specific functions. This makes it less generic than MPC or HE techniques but due to the application specific design, FE schemes are more efficient. Briefly, in an FE scheme, a user can encrypt the data using a public key and decryption returns the function calculated on the input plaintext. This allows the user to share their data in encrypted form and allow the third party (e.g., cloud server) to retrieve a function of it without revealing the original data.

Inner-product functional encryption (IPFE) [ABCP15] is one of the FE constructions that support the computation of inner-product between two vectors. Similar to the MPC and HE techniques, FE or IPFE techniques are also computationally demanding and improving the efficiency of IPFE schemes is still an open problem to be solved. A recent work [MKMS22] has tried to address this issue by designing an IPFE scheme based on Ring-Learning with Errors (RLWE) [LPR10]. The main advantage here is the relatively shorter keys and possibility of using faster number theoretic transform (NTT) [Pol71] based polynomial multiplications. This is more efficient compared to the LWE [Reg04] based constructions [ABCP15, ALS16] with larger keys and slower matrix vector multiplications. This work is also *first* to provide concrete instantiations of their scheme with different levels of security. In their implementation, for the medium security level, the IPFE decrypt function takes 17ms, which seems to be reasonably fast. However, as an example a machine learning classification task can easily take up to hundreds of decryptions which is not very efficient for practical deployment. We deliberate on this further in Sec. 4.1.

Our fundamental motivation behind this work was to investigate efficient alternatives to mainstream privacy-preserving COED techniques such as FE and HE. We found that the RLWE-IPFE [MKMS22] scheme suits for the greatest part of our purpose. In this paper, we present cuFE, the *first* parallel implementation of the above mentioned RLWE-IPFE scheme on a graphics processing unit (GPU). We exploit the different levels of parallelism in the RLWE-IPFE scheme for efficient implementation on massively parallel platforms like GPU. We show that the performance of FE can be accelerated significantly to bring it within the realm of practicality. This highly optimized implementation can be used to offload IPFE decryption to a cloud server, enabling high performance and privacy preserving machine learning applications. Since GPUs are widely adopted in many cloud computing services (e.g., Amazon Web Services (AWS) [AWS] and IBM cloud [IBM]), our proposed solution can be widely adopted by many applications. Further, cuFE also allows the GPU-accelerated IPFE encryption to be performed on the user side, which is useful to the machine learning applications that frequently update their data.

The contributions of this paper are summarized briefly below:

1. We analyze the performance bottlenecks and identify the opportunities in parallelizing various computational components. We explore different parallelization techniques (coarse-grain and fine-grain) to speed up the RLWE-IPFE scheme [MKMS22]. We are the first to show how these computational components can be efficiently mapped to different independent computing units in a parallel computing platform, which in our case is a GPU.

2. We describe techniques to effectively parallelize the Gaussian sampling process in a massively parallel architecture like GPU. This is also the first Gaussian sampler implemented on a GPU that achieves a very high sampling throughput. On a RTX2060 GPU, our GPU-accelerated Gaussian sampler can generate 131.26×10^6 samples per second, which is 4.29× faster than the AVX2 implementation. This high performance Gaussian sampler is used to speed-up one of the most time-consuming operations in the RLWE-IPFE [MKMS22] scheme. To the best of our knowledge, this is first such parallel implementation constant-time discrete Gaussian sampling. This can be of independent interest for other lattice-based cryptographic schemes that requires a discrete Gaussian sampler such as lattice-based signatures [DDLL13, FHK+20].

3. Number theoretic transform (NTT) is another computation intensive operation in the RLWE-IPFE [MKMS22]. Recently, Lee et al. [LH21] demonstrated an efficient technique to compute parallel NTT by combining the first two levels. However, their technique employed a static indexing pattern, which cannot be extended to cover more NTT levels. In this paper, we propose a novel dynamic indexing pattern to allow the level combination to be done on more levels. On a RTX2060 GPU, our GPU-accelerated NTT with length $n = 4096$ can calculate 73590 NTTs per second, which is 1.39× faster than the technique proposed by Lee et al. [LH21]

4. We propose a privacy-preserving SVM classification technique using RLWE-IPFE scheme as a suitable use case to validate the performance of the proposed cuFE. We further discuss different approaches to perform the SVM classification protected by the RLWE-IPFE scheme to extract maximum efficiency. The advanced vector extensions (AVX2) optimized version of the RLWE-IPFE [MKMS22] takes 23s to classify 100 data with 591 support vectors on a CPU. With cuFE, we only need 0.7s to complete the same classification task, which shows a 33× speed-up. Our proposed GPU-accelerated solution opens up the possibilities to employ RLWE-IPFE [MKMS22] on other more complicated machine learning techniques, which is arguably too slow to be practical when executed on a CPU. For instance, the convolution operations on convolutional neural network (CNN) can be performed through a series of inner-product. This allows cuFE to be used in protecting the privacy of CNN computation with practical performance.

Our implementation is available in the public domain: https://anonymous.4open.science/r/cuFE-A8BB

2 Background

2.1 General Purpose GPU

GPU is a very well known parallel platform to speed up computations. It was initially developed for graphics applications, but later on expanded to other applications, known as general purpose GPU (GPGPU). This is possible due to the availability of software development kit (SDK), such as CUDA [NVI22] and OpenCL [ope]. GPUs are very popular platform in fields like artificial intelligence and machine learning to speed up their respective protocols. Usage of GPUs in cryptography is not new or uncommon (albeit less popular). Some examples are cryptanalysis [BG12, LS19, DSvW21], speeding up cryptographic algorithms [JKA$^+$21, ABVMA18, LH21, Tez21], crypto-currency mining, etc. We have provided a brief overview of the GPU architecture and programming model below.

3 Overview of GPU Architecture

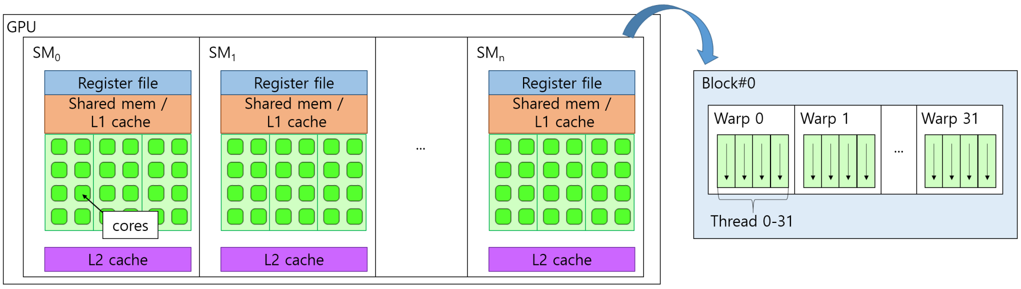


Figure 1: Overview of the SM architecture in a GPU

GPU is a massively parallel architecture, consists of hundreds to thousands of cores. Unlike the CPU core, GPU cores do not have complicated hardware to handle branch operations. However, a GPU consists of many independent cores; they are grouped into a larger unit called Streaming Multiprocessor (SM), which is illustrated in Figure 1. For instance, the RTX2060 is a GPU with Turing architecture that comes with 34 SMs, each SM consists of 64 cores. This massively parallel architecture allows GPU to be used for accelerating many computationally expensive algorithms. CUDA is the SDK released by NVIDIA to ease the programming of GPU for general purpose computing. Under the CUDA programming model, multiple threads are organized as a block, and multiple blocks form a grid. Referring to Figure 1, each thread and block can be indexed individually using the built-in variables (*threadIdx* and *blockIdx*). To allow efficient instruction scheduling, 32 threads are grouped into one warps, in which all threads within the warp execute the same instructions.

Besides the parallel processor architecture, GPUs also has a deep memory structure which is different from the CPU. The fastest memory in the GPU is the register, followed by shared memory and global memory (DRAM). Registers are very fast but small in size (e.g., 64K words per SM for the RTX2060). Shared memory is a user-managed cache with configurable sizes, ranging from 48K–128K words. On the other hand, global memory is large (several gigabytes) but very slow compared to registers and shared memory. When we use the GPU, data is usually transferred to the global memory first. GPU processes these data by moving them to the registers or shared memory whenever necessary. Many GPU implementation techniques center around the intelligent use of memory to achieve high throughput.

4 GPU Programming Model

There are two common strategies to map the computation of an algorithm to the GPU, based on different types of parallelism. Fine-grain parallelism refers to the inner parallelism

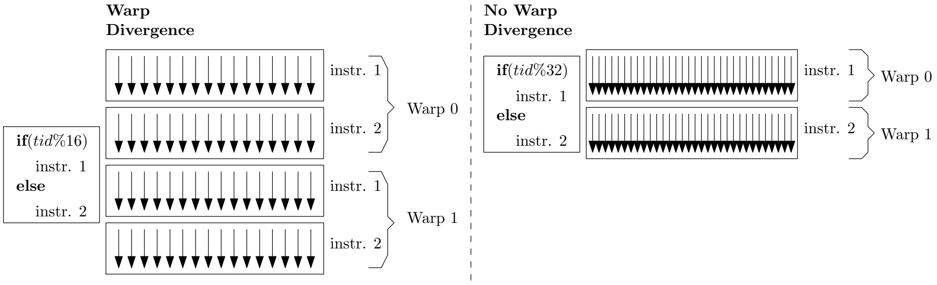


Figure 2: Warp divergence issue in the GPU.

that can be found in an algorithm. For instance, matrix-vector multiplication exhibits rich inner parallelism, which can be easily parallelized by multiple threads in a GPU. On the other hand, coarse-grain parallelism, also known as data-level parallelism, refers to the case where each parallel thread computes one instance of the target algorithm on different input data, in a serial manner. For example, one can instantiate multiple threads on a GPU, wherein each thread computes one AES encryption in serial. Fine-grain parallelism relies on the amount of parallelizable computations within an algorithm in order to achieve a short latency. However, some algorithms may not have sufficient parallelizable computations to fully exploit the GPU resources. Coarse-grain parallelism can be very useful in throughput oriented applications, but it may create a very long latency if the target algorithm is very complicated. Hence, a combination of fine-grain and coarse-grain parallelism could be useful for certain applications to achieve a balanced latency and throughput performance. A notable example can be found in [LH21], where the authors proposed to compute one Kyber [BDK+18] key-encapsulation mechanism (KEM) per GPU block (fine-grain), and utilize many parallel blocks (coarse-grain) to execute many different instantiations.

After deciding the strategy to map the computational tasks to a GPU, we need to make sure that our implementation always allow all the threads from the same warp to execute the same instruction. Note that the GPU schedules instructions in a lock-step of 32 threads (one warp), so that all threads within the same warp can execute the same instruction together. Failure to abide to this rule may cause the warp to re-schedule different instructions within the same warp, which has a significant penalty. This warp divergence issue is illustrated in Figure 2. The first *if/else* statement divides one warp into two parts, wherein 16 threads execute instruction 1, and the other 16 threads execute instruction 2. This causes each warp to issue two different instructions and complete them in two cycles. Note that when the first 16 threads are executing instruction 1, another half of the warp is actually idle. Since there are two diverged paths in this example, we name it as 2-way warp divergence. An inefficient implementation may have more than 2-way warp divergence, which can be a serious performance bottleneck. On the other hand, a divergence free implementation (the second *if/else* loop on the right) allows the entire warp to compute the same instruction in one clock cycle. This allows the GPU to schedule all instructions in a highly efficient manner.

4.1 Overview of RLWE-IPFE

Functional encryption is a generalization of traditional public-key encryption, which goes beyond the traditional all-or-nothing access to the data of public-key encryption. More formally for a particular functionality f, an authorized user holding a secret-key sk_f, and an encrypted message m, an FE scheme allows the authorized user to calculate $f(m)$ by applying the decryption operation of the FE using the key sk_f. An FE scheme has 4 operations namely `Setup, KeyGen, Encrypt,` and `Decrypt`. A `Setup` returns a public-key

(pk) and a master secret-key msk for a given security level. The `KeyGen` function accepts the msk and a function f and returns a secret-key sk_f associated with the function f. The `Encrypt` function accepts the public-key pk and a message m and returns the ciphertext ct_m. Finally, the `Decrypt` function accepts sk_f and ct_m and returns $f(m)$. In practice, a generic FE scheme is highly inefficient. Moreover, they are either secure against a bounded number of collusions [GVW12, GKP$^+$13] or dependent on strong primitives [GGH$^+$13]. Therefore an alternate research direction has emerged to design FE schemes with specific classes of functionalities which are efficient in practice. Among these functionalities inner-product is one of the most popular subclasses of FE.

In IPFE, the message is a vector $\mathbf{x} \in \mathcal{M}^l$, the secret-key $sk_{\mathbf{y}}$ of an user is associated with another l-dimensional vector $\mathbf{y}$. In IPFE scheme an user holding $sk_{\mathbf{y}}$ can compute $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{l} x_i * y_i$ i.e. the inner-product without revealing $\mathbf{x}$. IPFE has been instantiated using different hard problems such as decisional Diffie-Hellman [ABCP15], decisional Composite Reminder [ALS16], and Learning With Errors (LWE) [ABCP15, ALS16]. Except for the LWE based instantiations, the other IPFE constructions are vulnerable to quantum attacks. For LWE-based IPFE schemes, the schemes are computationally demanding with very large keys. Hence it was an open problem to design a quantum-secure and efficient IPFE scheme. The RLWE-IPFE scheme proposed by Mera et al. [MKMS22] is first such scheme which provides a solution to this problem. The authors have used hard lattices problems on ideal lattices (RLWE) instead of standard lattices (LWE) as the underlying hard problem. This has helped them to design a scheme with smaller key-sizes and faster execution due to the use of NTT-based multiplication. To prove their claim, the authors also provided concrete parameters for their scheme for different levels of security as shown in Table. 1 and implementations. We briefly describe the single input FE scheme from Mera et al. [MKMS22] below. For more detailed information we refer the interested reader to the original paper [MKMS22].

4.1.1 Construction

We define the polynomial ring $R = \mathbb{Z}[x]/x^n + 1$ and the quotient ring $R_q = R/qR = \mathbb{Z}_q[x]/x^n + 1$. 1_R is the identity element of this ring where the constant term is set to 1 and all other coefficients are set to 0. The RLWE-IPFE scheme allows encrypting non-negative vectors. Further, the l-dimensional message $\mathbf{x}$ and the vector $\mathbf{y}$ are bounded by B_x and B_y such that $||\mathbf{x}||_\infty \leq B_x$ and $||\mathbf{y}||_\infty \leq B_y$. K is a constant set to be greater than the maximum value of the inner-product i.e. $K > lB_xB_y$. D_σ is a discrete Gaussian distribution with standard deviation σ and mean 0. When a variable v is sampled randomly from a particular distribution χ we write $a \leftarrow \chi$. This notation is extended to polynomial or vectors to denote that each vector elements or coefficients have been sampled independently from a particular distribution. Also, $[l]$ stands for the set $\{1, \ldots, l\}$.

Setup

1. Sample $a \in R_q$ uniformly at random
2. Sample $(s_i, e_i \leftarrow D_{\sigma_1}) \in R$ for $i \in \{1, 2, \cdots, l\}$
3. Compute $pk_i = a \cdot s_i + e_i \in R_q$ for $i \in \{1, 2, \cdots, l\}$
4. Set, $msk = \{s_i \mid i \in [\ell]\}$ and $mpk = \big(a, \{pk_i \mid i \in [l]\}\big)$

Encrypt Given a message $\mathbf{x} = (x_1, x_2, \cdots, x_\ell) \in \mathbb{Z}^l$ and $||\mathbf{x}||_\infty \leq B_x$, the `Encrypt` is as follows,

- Sample $(r, f_0 \leftarrow D_{\sigma_2}) \in R_q$
- Sample $(f_i \leftarrow D_{\sigma_3}) \in R_q$ independently for all $i \in \{1, 2, \cdots, l\}$

- Calculate $ct_0 = a \cdot r + f_0,\ \ ct_i = pk_i \cdot r + f_i + \lfloor q/K \rfloor x_i \cdot 1_R$ for all $i \in [l]$
- Output $ct_x = \big(ct_0, \{ct_i \mid i \in [l]\}\big)$ as encryption of $\mathbf{x}$.

KeyGen Given a vector $\mathbf{y} = (y_1, y_2, \cdots, y_l) \in \mathbb{Z}^l$ such that $||y||_\infty \leq B_y$, The decryption-key corresponding $\mathbf{y}$ is calculated as below.

$$sk_{\mathbf{y}} = \sum_{i=1}^{l} y_i s_i \in R$$

Decrypt To decrypt the ciphertext $ct_x = \big(ct_0, \{ct_i \mid i \in [l]\}\big)$ using the decryption key sk and $\mathbf{y}$ we compute,

$$d = \left(\sum_{i=1}^{l} y_i ct_i\right) - ct_0 \cdot sk_{\mathbf{y}}$$

This d will be very close to $\lfloor q/K \rfloor \langle x, y \rangle 1_R$ with some noise. For properly chosen parameters these noise can be eliminated and we can extract $\langle x, y \rangle$. This is discussed below.

Correctness

We can write the decryption value d as

$$d = \left(\sum_{i=1}^{l} y_i ct_i\right) - ct_0 \cdot sk_{\mathbf{y}} = \underbrace{\sum_{i=1}^{l}(y_i e_i r + y_i f_i - y_i s_i f_0)}_{\texttt{noise}} + \lfloor q/K \rfloor \langle x, y \rangle 1_R$$

To recover the inner-product, we need $||\texttt{noise}||_\infty < \lfloor q/2K \rfloor$. We fix a security parameter κ. With non-negligible probability we have, $||e_i||_\infty, ||s_i||_\infty \leq \sqrt{\kappa}\sigma_1$, also $||r||_\infty, ||f_0||_\infty \leq \sqrt{\kappa}\sigma_2$ and $||f_i||_\infty \leq \sqrt{\kappa}\sigma_3$. Therefore, we can bound the noise as,

$$\left|\left|\sum_{i=1}^{\ell}(y_i e_i r + y_i f_i - y_i s_i f_0)\right|\right|_\infty \leq \ell(2n\kappa\sigma_1\sigma_2 + \sqrt{\kappa}\sigma_3)B_y$$

Hence, for correctness, we need to choose the parameters of the RLWE-IPFE scheme such that $l(2n\kappa\sigma_1\sigma_2 + \sqrt{\kappa}\sigma_3)B_y < \lfloor q/2K \rfloor$. FE or specifically IPFE can have different variants such as multi-client FE which is a stronger form of FE where there is multiple sources of the data comes. And it is possible for each client to encrypt their data individually without thrusting other clients [CSG+18]. Another example is decentralized multi-client FE [ABKW19, CSG+18] which eliminates the need for a trusted authority who maintains all the secret keys in the system in and generates the decryption keys.

We want to stress that, similar to the implementations of Mera et al. [MKMS22], we have considered the single input FE scheme here to demonstrate our GPU implementation. Nevertheless we strongly believe that our optimization techniques can be extended to other variants of RLWE-IPFE trivially as the basic computational elements remain the same for different variants of RLWE-IPFE. Henceforth, we use the term RLWE-IPFE for the IPFE scheme by Mera et al. [MKMS22] only.

4.2 Related Work

FE can provide privacy preservation on artificial intelligence (AI) applications. For example, Bahadori et al. [BJMS21] and Ligier et al. [LCFS17] presented two interesting works that utilized FE for Quadratic Functions (FE-QF) and IPFE respectively, to protect data classification on MNIST dataset. A similar work was also presented by Ryffel et

Table 1: Parameter sets of RLWE-IPFE [MKMS22].

Security level	PQ Security	FE Bounds	Gaussian Parameters	Ring Parameters	CRT moduli
Low	76.3	$B_x : 2$ $B_y : 2$ $l = 64$	$\sigma_1 : 33$ $\sigma_2 : 59473921$ $\sigma_3 : 118947840$	n : 2048 $\lceil log\ q \rceil : 66$	$q_1 : 2^{14} - 2^{12} + 1$ $q_2 : 2^{23} - 2^{17} + 1$ $q_3 : 2^{29} - 2^{18} + 1$
Medium	119.2	$B_x : 4$ $B_y : 16$ $l = 785$	$\sigma_1 : 225.14$ $\sigma_2 : 258376412.19$ $\sigma_3 : 516752822.39$	n : 4096 $\lceil log\ q \rceil : 86$	$q_1 : 2^{24} - 2^{14} + 1$ $q_2 : 2^{31} - 2^{17} + 1$ $q_3 : 2^{31} - 2^{24} + 1$
High	246.2	$B_x : 32$ $B_y : 32$ $l = 1024$	$\sigma_1 : 2049$ $\sigma_2 : 5371330561$ $\sigma_3 : 10742661120$	n : 8192 $\lceil log\ q \rceil : 101$	$q_1 : 2^{17} - 2^{14} + 1$ $q_2 : 2^{20} - 2^{14} + 1$ $q_3 : 2^{32} - 2^{20} + 1$ $q_4 : 2^{32} - 2^{30} + 1$

al. [RPB+19], where FE was used to build a secure deep learning framework. Another recent work from Mera et al. [MKMS22] also demonstrated the secure classification task using logistic regression on MNIST dataset. Besides protecting the classical AI algorithms, IPFE can also be used to safeguard more advanced AI algorithm like federated learning [YFX+21]. On the other hand, IPFE is also used to protect outsourced computations on encrypted database against the untrusted cloud server [YSQW20]. Although IPFE is getting popular, the performance presented in previous work is still not very impressive, due to the its inherent heavy computations.

Number theoretic transform (NTT) is the most time-consuming operation in RLWE-IPFE [MKMS22]. Accelerating the computation of NTT remains an active research area that attracted a lot of attention over the past decade. Pöppelmann et al. [POG15] proposed to combine the Cooley-Tukey (forward) [CT65] and the Gentleman-Sande (inverse) NTT to remove the bit-reverse step. Mera et al. applied this strategy [POG15] into their RLWE-IPFE [MKMS22] to achieve a better performance. The interest to utilize specialized hardware to accelerate NTT is growing in recent years. For instance, Fritzmann et al. [FS19] designed a hardware architecture with low-power consumption, which was experimentally verified on an ASIC platform. Zhang et al. [ZYC+20] also presented an optimized NTT implementation on an FPGA to support the Newhope key-encapsulation mechanism (KEM). However, these techniques are specific to the specialized hardware platforms (ASIC and FPGA), which are not commonly found in the cloud server environment.

GPUs are becoming the de facto accelerator in many cloud services like AWS [AWS] and IBM [IBM]. Some previous research works were performed to investigate the efficient ways to implement NTT on a GPU. Recently, Lee et al. [LH21] presented the NTT implementation on GPUs for Kyber KEM [BDK+18]. They proposed to combine the first two levels of NTT so that more computations can be performed on the registers, effectively reducing the accesses to shared memory. However, each block in a GPU can only host maximum 1024 threads, so the technique proposed by Lee et al. [LH21] is only suitable when the polynomial length $n \leq 2048$. Another recent work by Ozerk et al. [ÖEM+22] presented a technique to compute NTT with $n \geq 2048$ by computing multiple butterfly operations per thread. This technique does not require synchronization across different blocks, but it is not a fully parallel solution. Another interesting work by Jung et al. [JKA+21] shows an optimized implementation of NTT for fully homomorphic encryption (FHE). However, FHE employs NTT with very large polynomial length ($n = 65536$), which is much larger than the target RLWE-IPFE ($2048 \leq n \leq 8192$). Note that the available GPU optimization strategies are directly affected by the polynomial length. This implies that the techniques proposed by Jung et al. [JKA+21] may not be directly applicable to RLWE-IPFE.

Table 2: Computational breakdown of the RLWE-IPFE (medium security level) [MKMS22]

	Gaussian Sampling	Polynomial Multiplication	Scalar Multiplication	CRT	Others[1]
Setup	28%	77%	–	–	5%
Encrypt	23%	73%	–	–	4%
KeyGen	–	–	98%	1%	1%
Decrypt	–	6%	91%	2%	1%

[1] Including random seeds, point-wise addition/subtraction and etc.

Algorithm 1: Pseudocode: Scalar multiplication in KeyGen and Decrypt.

```
Input: Vector $x$ with length $n$, Matrix $y$ with size $l \times N_{mod} \times n$
Output: Vector $z$ with length $n$
1 for $i$=0; $i$< $l$; $i$=$i$+1 do
2     for $j$=0; $j$< $N_{mod}$; $j$=$j$+1 do
3         for $k$=0; $k$< $n$; $k$=$k$+1 do
4             $mac = x[j][i] \times y[i][j][k]$;
5             $z[j][k] = mac + z[j][k]$;
```

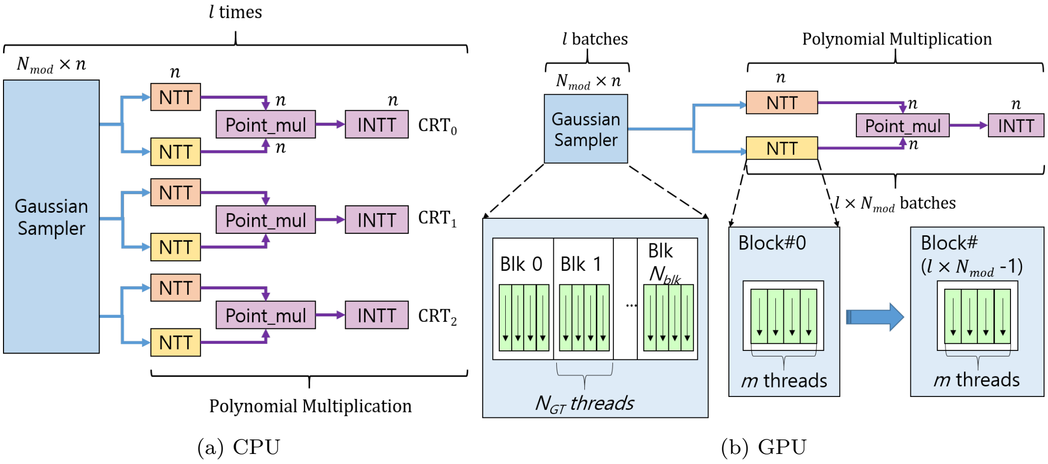
5 Proposed method

In this section, we first identify the main computational bottlenecks and the parts of of RLWE-IPFE implementation which can be accelerated using parallelization. Then, we describe our strategies to parallelize the RLWE-IPFE technique on GPU platforms.

5.1 Bottlenecks and Scopes for Parallelization

Table 2 shows the breakdown of the major computations in the reference implementation of RLWE-IPFE [MKMS22], evaluated on an Intel i7-9700F processor. It can be seen that polynomial multiplication using NTT is the most time-consuming operation, taking up to 77% and 73% of the `Setup` and `Encrypt` time respectively. Gaussian sampling is the next time-consuming operation, which is only used in `Setup` and `Encrypt`. The remaining operations are relatively lightweight compared to Gaussian sampling and polynomial multiplication. Note that in the `KeyGen` and `Decrypt` process, majority of time is spent in computing scalar multiplication, which can be easily parallelized on a GPU. However, the Gaussian sampling and NTT requires significant effort to fully harnessing the massively parallel GPU architecture.

Algorithm 1 shows the scalar multiplication found in `KeyGen` and `Decrypt` functions. The inputs are a vector with length n and a matrix with size $l \times N_{mod} \times n$. Scalar multiplication is performed between a point from the vector x and a vector extracted from matrix y, and the results are added with the previous results from vector z (lines 3 – 5). Since there are N_{mod} CRT channels, scalar multiplication is performed N_{mod} times (line 2). This process is repeated for l times (line 1). Althogh the scalar multiplications is highly parallelizable, a naive implementation on a GPU may lead to sub-optimal performance. For instance, one can instantiate l blocks and m threads, where each thread is assigned to compute the j and k loops (lines 2 – 5). This approach is easy to implement, but suffers from potential data hazard that happens when each block is accessing the vector z in line 5. To avoid this problem, we need to synchronize all thread blocks after each MAC operation, which is very costly. In this paper we propose to parallelize the k loop (line 3) across GPU blocks. We instantiate 1024 threads (the maximum threads per GPU block) and $n/1024$



(a) CPU (b) GPU

Figure 3: Mapping the implementation of Gaussian sampler and polynomial multiplication to GPU

blocks to perform the MAC operations in lines 3 – 5. In this way, each thread is operating on a different element in the vector z over the i and j loops. This avoids the data hazard issue and do not needs any block-wise synchronization, which is very efficient on the GPU.

Referring to Figure 3(a), the `Setup` and `Encrypt` in RLWE-IPFE involve the Gaussian sampling and polynomial multiplication, which are the most time-consuming computations. The Gaussian Sampler generates $N_{mod} \times n$ samples which are split into three Chinese Remainder Theorem (CRT) channels. The subsequent polynomial multiplication involves NTT, point-wise multiplication and inverse NTT. This process is repeated for l times. A close observation reveals that the NTT and inverse NTT exhibit rich parallelism as they are operating in polynomials with length n. Referring to Figure 3(b), NTT and inverse NTT can be easily mapped to a GPU block with m parallel threads, and repeats n/m rounds to complete the computation. The polynomial multiplication is performed l times on different random samples that are independent of each other. Hence, we can utilize l GPU blocks to compute all the polynomial multiplication in parallel. To achieve this, we need to ensure that the Gaussian samples are ready before all polynomial multiplication begins. Hence, the Gaussian Sampler needs to generate $l \times n$ samples first, followed by the parallel polynomial multiplication. The Gaussian sampling process can be parallelized in a coarse-grain manner by utilizing N_{blk} blocks and N_{GT} threads, in which each thread computes a few Gaussian samples.

In the subsequent sub-sections, we present the detailed explanation on the proposed parallel Gaussian sampling and optimized NTT implementation techniques on the GPU.

5.2 High Performance Parallel Gaussian Sampling

5.2.1 Parallel Discrete Gaussian Sampler

As described in Sec. 4.1, the RLWE-IPFE needs to generate lots of samples from 3 different discrete Gaussian distributions namely σ_1, σ_2, and σ_3 (Table. 1). This is quite different compared to other lattice-based schemes [DDLL13, FHK+20] where discrete Gaussian sampling is used in mainly two different ways. Firstly, in the RLWE-IPFE scheme the standard deviations of the discrete Gaussian distributions (σ_2, σ_3) are orders of magnitude larger compared to other lattice-based schemes such as Falcon (2 or $\sqrt{5}$) and BLISS (< 300) digital signature schemes. Secondly, in our current scenario we need to generate samples from 3 different distributions whereas in other lattice-based schemes there is only one distribution. We also have to keep in mind that, since the samples from the discrete Gaussian distributions are secret values, we have to execute the sampling operation in constant-time. Otherwise, it is potentially feasible to mount efficient side-channel attacks [GBHLY16, Pes16] using timing leakage. Although, there exists some techniques [KLL21, HKR$^+$18, HPRR20, KRR$^+$18, ZSS19, KRVV19] for constant-time discrete Gaussian sampling, the method FACCT [ZSS19] proposed by Zhao et al. is most suitable for our scenario. For this method to generate a sample from a discrete Gaussian distribution with standard deviation σ_t, first a sample $x \leftarrow D_{\sigma_0}$ from a smaller base distribution with fixed $\sigma_0 = \sqrt{1/(2\ln 2)}$ is generated. Next, another integer y is sampled uniformly from the set $\{0, 1, \ldots, k-1\}$ where $k = \lceil \sigma_t/\sigma_0 \rceil$. Finally, a sample $z = y + kx$ is produced from D_{σ_t} with probability $\rho = \exp\big(\frac{-y(y+2kx)}{2\sigma_t^2}\big)$. The last step of this method is performed using a Bernoulli sampler first proposed in the BLISS signature scheme [DDLL13]. This method follows the distribution D_{σ_t} [ZSS19]. Also, if the Bernoulli and the base sampling can be done in constant-time then the whole process runs in constant-time. FACCT [ZSS19] introduced a constant-time Bernoulli sampler (a non-constant-time version was proposed in BLISS) in their work. For the sampling from the base sampler, it is best suited to use the linear search cumulative distribution table (CDT) based sampler [BCNS15]. Since σ_0 is small, the CDT table is very small (12 entries), the linear search is very efficient and simpler to implement. In this case, the bit-sliced based sampling methods which has some pre- and post-processing costs do not offer any discernible advantage. As the most complex part of this method which is sampling from the base sampler D_{σ_0} is fixed, we can generate samples from different distributions D_{σ_1}, D_{σ_2}, and D_{σ_3} by varying the constant k only. This results in a very simple and efficient solution for the constant-time discrete Gaussian sampling for the RLWE-IPFE scheme.

One way to implement this technique on a GPU is to exploit the fine-grain parallelism within each Gaussian sampling process. For instance, the AVX2 impelemtntation of FACCT [ZSS19] can generate four random samples in parallel. The reference implementation provided by RLWE-IPFE scheme [MKMS22] follows this approach to speed up the Gaussian sampling process. However, this approach does not exhibit sufficient parallelism to fully exploit the massively parallel architecture in a GPU. Note that for the `KeyGen` and `Encrypt` processes, we need to generate $l \times n$ Gaussian samples. For instance, for a medium security level, $4096 \times 785 = 3,215,360$ Gaussian samples are required for each `Encrypt`. Note that a typical real world application may need to encrypt more than hundreds of data vectors, resulting in a huge amount of computations required, which makes Gaussian sampling one of the main bottlenecks for cuHE.

In this paper, we propose to parallelize FACCT in a coarse-grain manner, wherein each thread generates one Gaussian sample independently. This is to avoid the need for any synchronization among threads, which can be expensive in a GPU. Referring to Figure 4, l GPU blocks are instantiated, where each block contains N_{GT} threads. Each thread is responsible of encrypting $24 \times 16 = 384$ bytes of AES counter values in the first iteration ($z = 0$). For instance, thread T_0 encrypts the counter values 0–23, T_1 encrypts the counter values 24–47, and so on. Within each thread, the random samples generated from AES encryption are passed through Uniform, CDT and Bernoulli sampling to obtain one Gaussian sample. If the sample is rejected, a new sample is generated by increasing the counter value. Note that there are $N_{GT} \times N_{blk} \times k \times 24$ produced at one time, so this counter value is increased accordingly to avoid using the same counter with other threads. We also limit this trial by a parameter N_s, so that there will not be too many repeated trials in a particular thread. This can avoid some threads spending too much time waiting for other threads that have many repeated trials. Within each thread, this process is repeated for k iterations, where $k = n/N_{GT}$. This is to ensure that each GPU block can generate n Gaussian samples. In total, $l \times n$ Gaussian samples are generated.

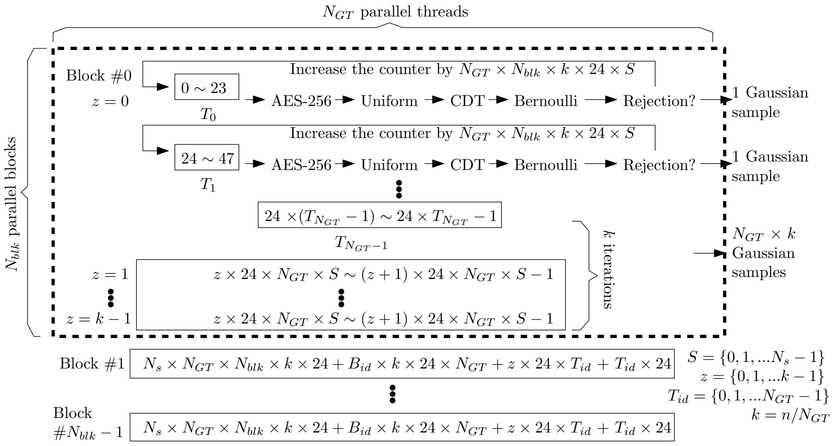

N_{GT} parallel threads
N_{blk} parallel blocks
Block #0
$z = 0$
Increase the counter by $N_{GT} \times N_{blk} \times k \times 24 \times S$
$0 \sim 23$
T_0
AES-256
Uniform
CDT
Bernoulli
Rejection?
1 Gaussian sample
Increase the counter by $N_{GT} \times N_{blk} \times k \times 24 \times S$
$24 \sim 47$
T_1
AES-256
Uniform
CDT
Bernoulli
Rejection?
1 Gaussian sample
$24 \times (T_{N_{GT}} - 1) \sim 24 \times T_{N_{GT}} - 1$
$T_{N_{GT}-1}$
k iterations
$N_{GT} \times k$ Gaussian samples
$z = 1$
$z \times 24 \times N_{GT} \times S \sim (z+1) \times 24 \times N_{GT} \times S - 1$
$z = k - 1$
$z \times 24 \times N_{GT} \times S \sim (z+1) \times 24 \times N_{GT} \times S - 1$
Block #1
$N_s \times N_{GT} \times N_{blk} \times k \times 24 + B_{id} \times k \times 24 \times N_{GT} + z \times 24 \times T_{id} + T_{id} \times 24$
Block $\#N_{blk} - 1$
$N_s \times N_{GT} \times N_{blk} \times k \times 24 + B_{id} \times k \times 24 \times N_{GT} + z \times 24 \times T_{id} + T_{id} \times 24$
$S = \{0, 1, ...N_s - 1\}$
$z = \{0, 1, ...k - 1\}$
$T_{id} = \{0, 1, ...N_{GT} - 1\}$
$k = n/N_{GT}$

Figure 4: Parallel Gaussian sampling on a GPU and the generation of counter values.

Table 3: Comparing the throughput of Gaussian sampling on CPU and GPU

	CPU[1] AVX2	GPU RTX 2060	Speed-up CPU / GPU
Throughput ($\times 10^6$ samples/s)	30.59	131.26	4.29×

[1] Intel i7-9700F processor

5.2.2 Other optimizations

AES-256 encryption is the most time-consuming part in Guassian sampling. A typical way to speed-up AES implementation on GPUs is to employ four pre-computed T-box [LCPG16] stored on the shared memory. A recent publication [Tez21] shows that using one T-box can eliminate all the shared memory bank conflicts and achieve significant speed-up compared to using four T-box. We follow the strategy described by [Tez21] in our AES-256 implementation using only one T-box. Note that since each thread is generating their own Gaussian samples independently, the sequence of these samples are stored differently compared to the serial version. It is not an issue since the sequence of Gaussian samples does not affect the security of the Gaussian sampler. However, due to this reason, the test vectors generated from the CPU serial version may not be same with cuFE. We do not store the random samples generated from AES-256 in shared memory, because it exceeds the maximum allowable shared memory size. Instead, we store the random samples in the local memory within each thread, which allows the computation to enjoy a better cache performance compared to global memory [NVI22]. The uniform, CDT and Bernoulli sampling operates on these random samples stored in the local memory. The CDT sampling table was stored in the constant memory as it is accessed globally by all threads. We have also tried to store this CDT table on the shared memory, but it does not provide any performance benefit. This is because the CDT sampling process is relatively lightweight compared to the other operations in Gaussian sampling. Moreover, loading this table onto the shared memory introduces some overhead, which potentially offsets its benefits. After all the Gaussian samples are generated, we move them from local memory to the global memory. Referring to Table 3, the proposed GPU-based Gaussian sampler is 4.29× faster than the AVX2 implementation.

5.3 Number Theoretic Transform

This subsection presents various techniques to optimize the parallel NTT implementation on GPUs for low and medium security RLWE-IPFE, where the polynomial lengths are

Algorithm 2: Pseudocode: Combining the first two levels in Kyber NTT [LH21]

```
   Input: Polynomial r with length $n = 256$, precomputed twiddle factors twi_tb
   Output: Polynomial r in the NTT domain
1  sel = blockIdx.x % $N_{mod}$;     // Select CRT channel through the block ID
2  level = 1; /* 64 threads are launched in parallel, where tid refers to
     the ID of each thread.  */
3  len = 128;  twi = twi_tb[sel][level + $\lfloor tid/len \rfloor$];
4  j1 = $\lfloor tid/len \rfloor$ * len + tid;
5  j2 = $\lfloor tid/len \rfloor$ * len + tid + 64;
6  t = mod_mul(twi, r[j1 + len]);                  // Modular multiplication
7  g1 = r[j1] - t;  g2 = r[j1] + t;                   // r[j1 + len]; r[j1];
8  t = mod_mul(twi, r[j2 + len]);
9  g3 = r[j2] - t;  g4 = r[j2] + t;                   // r[j2 + len]; r[j2];
10 level = level $\times$ 2;
11 len = 64;  twi = twi_tb[sel][level + $\lfloor tid/len \rfloor$];
12 j1 = $\lfloor tid/len \rfloor$ * len + tid;
13 j2 = $\lfloor tid/len \rfloor$ * len + tid + 128;
14 t = mod_mul(twi, g4);
15 r[j1 + len] = g2 - t;  r[j1] = g2 + t;             // r[j1 + len]; r[j1];
16 twi = twi_tb[sel][level + $\lfloor (tid + 64)/len \rfloor$];
17 t = mod_mul(twi, g3);
18 r[j2 + len] = g1 - t;  r[j2] = g1 + t;             // r[j2 + len]; r[j2];
19 level = level $\times$ 2;
20 for len=32; len$\geq$ 2; len=len/2 do
       /* Computes each level in parallel, without combining levels.  */
```

$n = 2048$ and $n = 4096$ respectively. Recall that the Gaussian samples generated are stored in the global memory. To perform the NTT and iNTT efficiently, these Gaussian samples are loaded onto shared memory for a faster access speed. After the polynomial multiplication completes, these data are written back to the global memory for subsequent computations. Recently, Lee et al. [LH21] proposed a two-level-combination technique to utilize more registers for computations. In this paper, we propose a novel technique that improves the this [LH21] further. In particular, the two-level-combination technique proposed by [LH21] only works for the first two levels of NTT. We propose a novel technique that extends this to cover all the levels in NTT computation. This allows more intermediate data to be stored in registers, effectively reducing accesses to the slower shared memory.

5.3.1 Extended Level-Combination in NTT

Algorithm 2 shows the NTT implementation by [LH21] targeting polynomials with $n = 256$. Lee et al. [LH21] only combined the first two levels in their NTT implementation, due to many inefficient *if/else* control statements to combine the remaining levels. Figure 5 shows the detailed data indexing pattern in NTT used in Algorithm 2. The registers $g1$–$g4$ are used to store the result of the butterfly operations when $len = 128$ (lines 6 and 8 in Algorithm 2). These registers $g1$–$g4$ are reused in the next level when $len = 64$. For instance, at $len = 128$, the data written to g2 and g4 (indices 0–127) are consumed by the same indices at $len = 64$. However, this technique is hard to extend to the subsequent levels when $len \leq 32$. Referring to Figure 5, the parallel threads are indexing a different register for butterfly operation on $len = 16$. The 0-15 and 32-47 threads access to $g2$ and $g4$, but the 16-31 and 48-63 threads access to $g1$ and $g3$. This requires an *if/else*

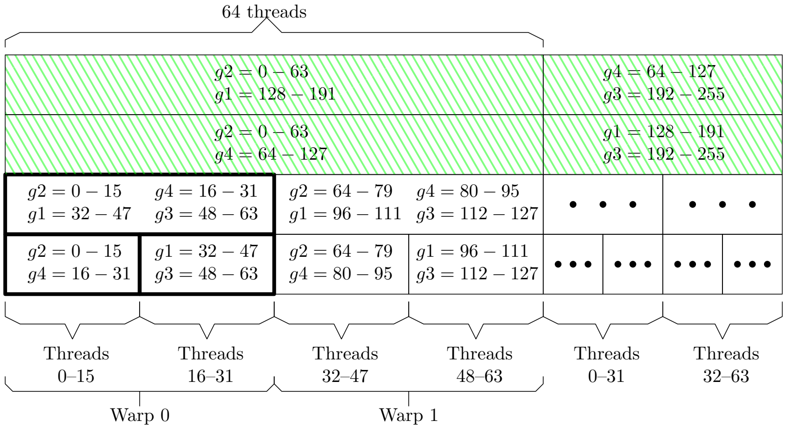


Figure 5: Data indexing patterns for level combination ($n = 256$) proposed by [[LH21]]

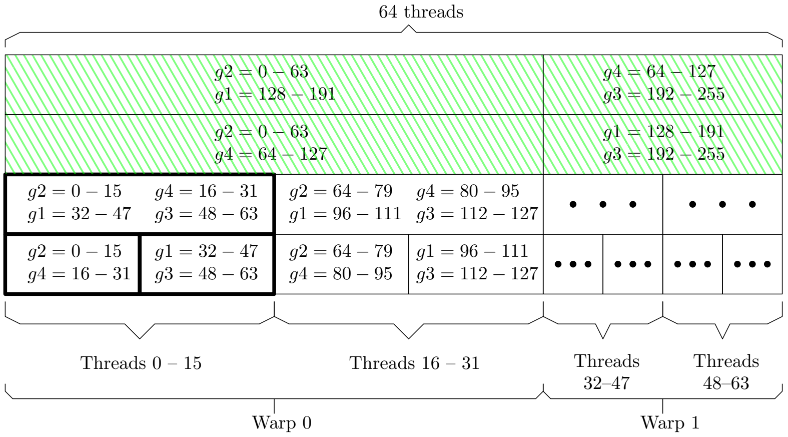


Figure 6: The proposed data indexing patterns for level combination ($n = 256$)

statement to separate each 16 threads so that they read the correct values. However, at level $len \leq 16$, the *if/else* control is performed within a warp (smaller than 32 threads), which creates warp divergence and seriously degrades the performance [LH21].

We propose a technique to extend this method to all levels in NTT, which is illustrated in Figure 6. To avoid using the *if/else* control statements, we need to ensure that all threads within a warp can read the same registers. Referring to Figure 6, the proposed indexing pattern uses only 16 threads to compute the first 32 NTT points, which is different from [LH21] that uses 32 threads (see Figure 5). This allows all threads within a warp to read the same registers, so we do not need the *if/else* control statement anymore. To achieve this efficient indexing pattern, we need dynamic indices for $len < 32$, because there is always four discontinuous ranges that changes dynamically for a different level when $len < 32$. The dynamic indices can be constructed by using the equations shown in Table 4. The term $\lfloor tid/len \rfloor \times 4 \times len$ is to construct the interval of the indices; $tid \ \% \ len$ is to avoid referring the same index; $len\times$ $\{0,\ 1,\ 2,\ 3\}$ is used to index register $g1 - g4$.

After successfully resolving the performance issues in levels $len < 32$, we can now extend the two-level combination technique [LH21] to all the levels in NTT. This two-level-combination technique (Algorithm 2, lines 3 – 17) is repeatedly applied to all levels, which is detailed in Algorithm 3. There are $n/4$ threads launched, each thread needs to compute two butterfly operations. The variable *level* is determined first by multiplying four at each

Table 4: The required indices for computing NTT at level $len = 16$

Register name	Static indices [LH21] (len = 16)	Dynamic indices (Ours) (all lengths)
g1	16-31, 80- 95, 144-159, 208-223	$\lfloor tid/len \rfloor \times 4 \times len + tid \ \% \ len + len \times 1$
g2	0-15, 64- 79, 128-143, 192-207	$\lfloor tid/len \rfloor \times 4 \times len + tid \ \% \ len + len \times 0$
g3	48-63, 112-127, 176-191, 240-255	$\lfloor tid/len \rfloor \times 4 \times len + tid \ \% \ len + len \times 3$
g4	32-47, 96-111, 160-175, 224-239	$\lfloor tid/len \rfloor \times 4 \times len + tid \ \% \ len + len \times 2$

Algorithm 3: Applying two-level-combination to all levels in NTT

```
   Input: Polynomial $r$ with length $n$, precomputed twiddle factors $twi\_tb$
   Output: Polynomial $r$ in the NTT domain.
   /* $n/4$ threads are launched in parallel, where tid refers to the ID
      of each thread.  */
1  $sel = blockIdx.x \ \% \ N_{mod}$;      // Select CRT channel through the block ID
2  for $level$=1; $level \leq n/4$; do
3      $level = level \times 4$;  $len = \lfloor n/level \rfloor$;
4      $j = \lfloor tid/len \rfloor \times 4 \times len + tid \ \% \ len$;
5      $twi = twi\_tb[sel][\lfloor level/4 \rfloor + \lfloor j/len/4 \rfloor]$;
6      $j1 = j$;  $j2 = j + len$;
7      $t$ = mod_mul($twi$, $r[j1 + len \times 2]$);
8      $g1 = r[j1] - t$;  $g2 = r[j1] + t$;             // r[j1 + len × 2] ; r[j1] ;
9      $t$ = mod_mul($twi$, $r[j2 + len \times 2]$);
10     $g3 = r[j2] - t$;  $g4 = r[j2] + t$;             // r[j2 + len × 2] ; r[j2] ;
11     $twi = twi\_tb[sel][\lfloor level/2 \rfloor + \lfloor j/len/2 \rfloor]$;  $j1 = j$;
12     $t$ = mod_mul($twi$, $g4$);
13     $r[j1 + len] = g2 - t$;  $r[j1] = g2 + t$                // r[j1 + len] ; r[j1] ;
14     $twi = twi\_tb[sel][\lfloor level/2 \rfloor + \lfloor j/len/2 \rfloor + 1]$;  $j2 = j + len \times 2$;
15     $t$ = mod_mul($twi$, $g3$);
16     $r[j2 + len] = g1 - t$;  $r[j2] = g1 + t$                // r[j2 + len] ; r[j2] ;
17 if $level < n$;                                           // for odd level
18  then
19     $level = level \times 2$;  $len = \lfloor n/level \rfloor$;
20     $j = \lfloor tid/len \rfloor \times 2 \times len + tid \ \% \ len$;
21     $twi = twi\_tb[sel][\lfloor level/2 \rfloor + \lfloor j/len/2 \rfloor]$;
22     $t$ = mod_mul($twi$, $r[j + len]$);
23     $r[j + len] = r[j] - t$;  $r[j] = r[j] + t$               // r[j + len] ; r[j] ;
24     $j = j + n / 2$;
25     $twi = twi\_tb[sel][\lfloor level/2 \rfloor + \lfloor j/len/2 \rfloor]$;
26     $t$ = mod_mul($twi$, $r[j + len]$);
27     $r[j + len] = r[j] - t$;  $r[j] = r[j] + t$;              // r[j + len] ; r[j] ;
```

two-level-combination (line 3), followed by the calculation of len (line 3). The variables j and len are used to calculate the indices to array r that stores the intermediate results. We observed that j and len depend on $level$, which allows the indices to array r to be calculated by the same formula at every level. For instance, len is calculated $\lfloor n/level \rfloor$ and j is calculated $\lfloor tid/len \rfloor \times 2^{\alpha} \times len + tid \ \% \ len$, which α is the number of combining levels. Line 5 loads the twiddle factors and lines 6 computes the indices $j1$ and $j2$. Lines 7–16 are modified from Algorithm 2, which combines the first two levels in NTT. The

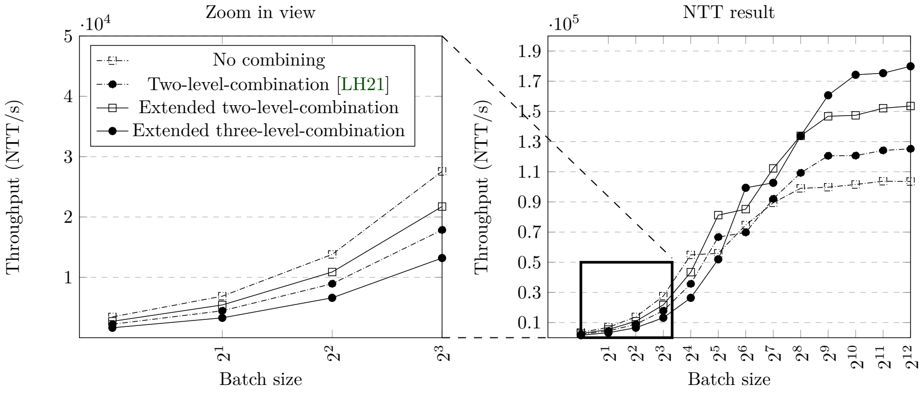


Figure 7: Comparing the implementation of NTT (n=2048)

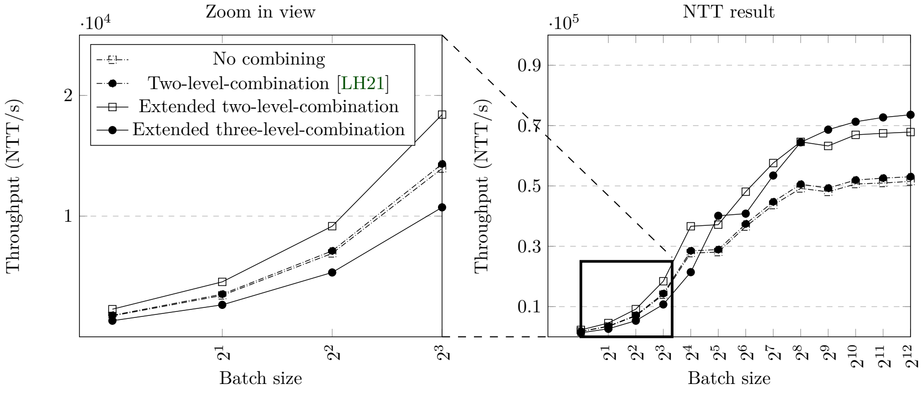


Figure 8: Comparing the implementation of NTT (n=4096)

intermediate results are stored back to the shared memory in lines 13 and 16. Note that the same two-level-combination (lines 2 – 16) is repeated for the subsequent levels. To handle the case where the number of levels is odd (e.g., $n = 2048$ with 11 levels), the last level processed lines 17–27. This allows us to cache some intermediate data in the registers and reduce the accesses to shared memory. Algorithm 3 can also be extended to support k-level-combination. For instance, to use three-level combination, we set $\alpha = 3$ and utilizes 2^3 registers, which computes 2^{3-1} butterfly operations in each round.

5.3.2 Micro-benchmarking the NTT Implementation

To evaluate the effectiveness of the proposed NTT implementation technique, we perform a micro-benchmark experiment and compare it to the technique proposed by Lee et al. [LH21]. In this experiment, we evaluate throughput of computing NTT with length of $n = 2048$ (low security level) and $n = 4096$ (medium security level), with batch sizes ranging from 1 – 4096. Referring to Figure 7, we observed that the implementation without any level combination achieve the best performance when the batch size is small (≤ 16). This is because the shared memory is not fully stressed when the batch size is small, so the effect of using more registers through level-combination technique is not obvious. However, when the batch size increases beyond 16, the two-level-combination technique [LH21] is giving a better performance. The proposed extended level combination techniques consistently outperforming [LH21] when the batch size increases beyond 32. Note that for large batch sizes (≥ 512), the proposed extended three-level-combination technique is clearly outperforming the other techniques. This is because combining three levels grants more opportunities to reuse the registers, compared to only combining two levels. The proposed technique is also better than [LH21] because the level-combination can be extended to cover all NTT levels. Similar performance can also be observed in Figure 8 for the case $n = 4096$. For applications that need to process a large batch of data, the extended three-level-combination is more suitable to be used. Note that we do not present the results for four-level-combination as it does not improve the performance further. In general, we can conclude that combining more NTT levels can allow a better throughput performance, but this performance gain is also limited by the available GPU resources. When a GPU is already achieving the peak performance, combining more NTT levels may not improve the performance anymore. The most suitable level to be combined using our technique varies across different GPUs, which can be determined experimentally.

5.4 Other Operations

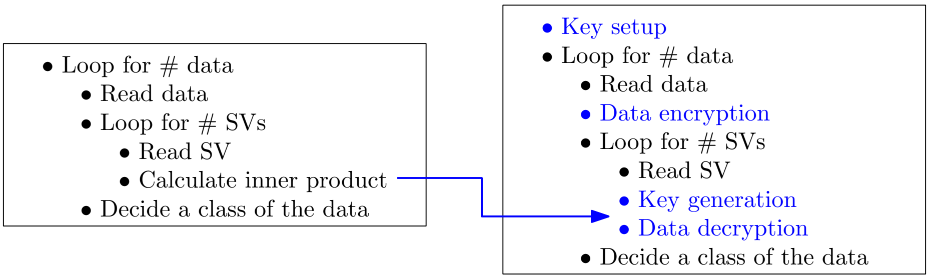
Besides scalar multiplication, Gaussian sampling, and polynomial multiplication, the other operations in RLWE-IPFE [MKMS22], have a negligible impact on the overall performance. Therefore, they can be implemented in a more straightforward manner. Such operations include random seed generation, CRT, inverse CRT, point-wise addition and subtraction. The random seeds are generated on the CPU as it is simple and it acquires the randomness from the operating system. The random polynomial used in `Setup` is also generated on the CPU, as this is only used once and the operations involved are not heavy. The iCRT involves computations in large integer, which is not trivial to parallelize in GPU. Since iCRT is also not consuming a long time, we execute the iCRT and final extraction of result on the CPU, using GMP library [GMP]. The other operations are all parallelized and executed on the GPU. For all these operations implemented on the GPU, the computations are performed on global memory instead of shared memory. This is because transferring data between global and shared memory introduces overhead. Since these operations are lightweight, the benefits of using shared memory is not significant. In Section 6.3, we also show a technique to merge some of the kernels, so that the data can completely reside in shared memory, resulting in a slightly better performance.

6 Support Vector Machine

Support Vector Machine (SVM) is one of the most widely used supervised learning techniques for classification tasks. Given a set of training examples, SVM can be trained to make a binary linear classification. In particular, SVM kernel first maps the training examples to the points in the higher dimensional space. Then, it finds a line to classify the new input data into two categories, wherein the points nearer to the line are called Support Vectors (SVs). In this paper, we demonstrate a practical use case of SVM classification secured by the proposed cuFE. The SVM implementation is based on the popular open source library, libsvm [CL11] and the dataset ‘a1a’ from [FCLJ05].

6.1 Naive Solution: Exploiting the Fine-Grain Parallelism in IPFE

Figure 9 shows an overview of the steps in performing SVM classification in [CL11]. The unprotected SVM (left side) reads one data and an SV, then computes the SVM_kernel, which is essentially an inner product of data and the SV. This process is repeated for all the trained SVs, followed by a classification to determine the class of the input data. These three steps are repeated for all the data to be classified. Finally, SVM evaluates the accuracy of classification. The right side shows a baseline implementation of IPFE-protected SVM. The additional steps that enables IPFE protection include `Setup`, `KeyGen`,



SVM

Naïve SVM with IPFE

Figure 9: The overview of major steps in SVM [CL11]

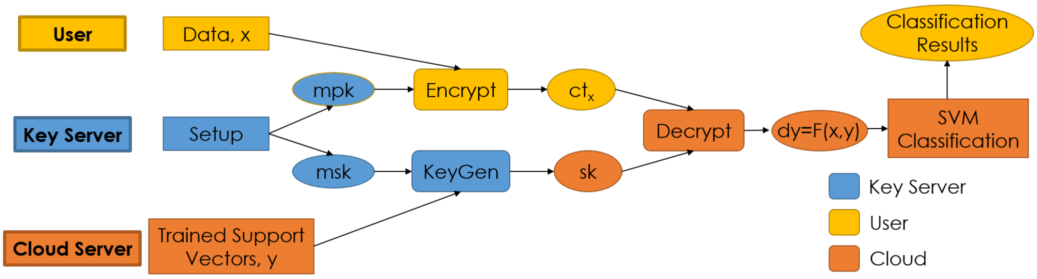


Figure 10: SVM classification with RLWE-IPFE

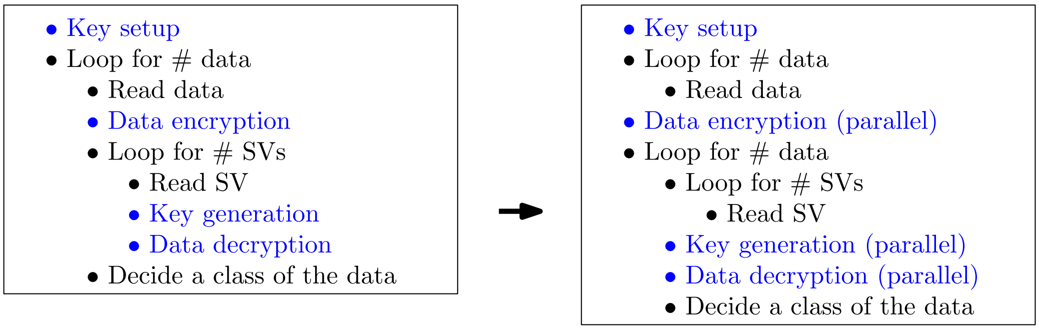
`Encrypt`, and `Decrypt`. This allows us to apply cuFE to SVM tasks, where the IPFE functions are parallelized and executed on the GPU.

Referring to Figure 10, the key server first performs `Setup` to generate the master private (*msk*) and public (*mpk*) keys. For the dataset that we have used in our experiment [FCLJ05], the data was stored in binary form. The dataset consists of a list of features values which is stored in the {*index*:*value*} pair. For example, the list {5:1, 7:1, ...} refers to a single vector of {0 0 0 0 1 0 1...}. The client first reads the data to be classified and expand the original binary data into a series of vectors (x). Next, the client obtains the *mpk* and encrypts the vectors, then send the ciphertext (ct_x) to the server. On the server side, the cloud server sends the trained SV (y) to the key server to obtain the decryption key (*Sk*), and uses this key to decrypt the result of inner product ($F(x, y)$) between the input data and SV. Finally, the results of all of SVs are used to perform the SVM classification. This process is repeated to classify all the input data.

The naive implementation only exploit the inner parallelism exists in RLWE-IPFE [MKMS22]. As an example, Algorithm 4 shows the `Decrypt` function for our naive SVM implementation. This algorithm receives one encrypted data ct_x, an SV y and the corresponding decryption key *sk* as input, which are copied to the GPU (line 1). The subsequent operations (lines 2 – 8) are all performed on the GPU based on our proposed cuFE. Note that the triple bracket $<<< X, Y >>>$ refers to launching X blocks and Y threads on a GPU. The intermediate result is copied back to the CPU (line 9) to extract the final result (line 10). `Encrypt` and `KeyGen` can be parallelized in the same manner using the proposed cuFE implementation. This approach exploits the inner parallelism within all the IPFE functions and provides a modest speed-up against the AVX2 implementation. However, it does not fully utilize the capability of a GPU, because fine-grain parallelism itself cannot generate sufficient workload to fully utilize the computational resources available in a GPU.

Algorithm 4: Naive SVM with IPFE: Decrypt Function

```
Input: Encrypted data $ct_x$, Support Vector $y$, Secret Key $sk$
Output: Inner Product $dy$
/* $d\_ycrt$ and $d\_c0sy$ are the intermediate results. */
1  Copy $ct_x, y$ and $sk$ from CPU memory to GPU global memory, denoted as
   $d\_ct_x, d\_array$ and $d\_sk$ respectively.
2  crt_convert_generic<<<$N_{mod}$, $l$>>>($d\_array$, $d\_ycrt$)
3  crt_mul_acc<<< $\lfloor n/1024 \rfloor$, 1024 >>>($d\_ct_x, d\_ycrt, dev\_dy$)
4  NTT<<<$N_{mod}$,1024>>>($d\_sk$)
5  NTT<<<$N_{mod}$, 1024>>>($d\_ct_x + l \times N_{mod} \times n$)
6  point_mul<<<$N_{mod}$, 1024>>>($d\_c0sy, d\_sk, d\_ct_x + l \times N_{mod} \times n$)
7  INTT<<<$N_{mod}$, 1024>>>($d\_c0sy$)
8  poly_sub_mod<<<$N_{mod}$, 1024>>>($dev\_dy$, $d\_c0sy$, $dev\_dy$)
9  Copy the intermediate result $dev\_dy$ from GPU global memory to CPU memory
   ($d\_y$).
10 round_extract_gmp($dy$, $d\_y$);
```



Naïve SVM with IPFE

Optimized SVM with IPFE

Figure 11: The summary of proposed SVM implementation.

6.2 Optimized Solution: Exploiting the Coarse-grain and Fine-grain Parallelism

To improve the performance of SVM classification, we propose to process multiple input data instead of one at a time. This coarse-grain parallel approach is described in Figure 11(b). `Encrypt` is executed only once after reading all the input data and processing them in a batch. Likewise, `KeyGen` and `Decrypt` processes are also executed in a batch after reading all SVs. Note that if the number of input data is too large to be computed within one batch, we can divide them into multiple batches, as long as it fits into the GPU memory. In this way, we are essentially exploiting the coarse-grain parallelism from SVM and the fine-grain parallelism from RLWE-IPFE [MKMS22], to speed-up the performance of cuFE on a GPU.

As an example, we present Algorithm 5 to demonstrate the proposed idea on the `Decrypt` process. Note that Algorithm 5 is similar to Algorithm 4, except that now we process more input data in a batch. This can be achieved by increasing the number of blocks (lines 2–4), where *batch_sz* defines the input size. If the number of input (SVs) is too large that a GPU cannot process them in a single batch, we can set *batch_sz* to a maximum size and repeat Algorithm 5 several times to fully process all input data. This technique is applicable to all other IPFE functions. By processing more input in a batch,

Algorithm 5: Optimized SVM with IPFE: Decrypt function

```
   Input: Encrypted data $ct_x$, Support Vector $y$, Secret Key $sk$, the number of input
          $batch\_sz$
   Output: Inner Products $dy$
   /* $d\_ycrt$ and $d\_c0sy$ are the intermediate results. */
1  Copy $ct_x, y$ and $sk$ from CPU memory to GPU global memory, denoted as
     $d\_ct_x, d\_array$ and $d\_sk$ respectively.
2  dim3 grid1($N_{mod}$, batch_sz)
3  dim3 grid2(n/1024, batch_sz)
4  crt_convert_generic<<<grid1, l>>>(d_array, d_ycrt)
5  crt_mul_acc<<<grid2, 1024>>>($d\_ct_x$, d_ycrt, dev_dy)
6  NTT<<<grid1, 1024>>>(d_sk)
7  NTT<<<grid1, 1024>>>($d\_ct_x + l \times N_{mod} \times n$)
8  point_mul<<<grid1, 1024>>>(d_c0sy, d_sk, $d\_ct_x + l \times N_{mod} \times n$)
9  INTT<<<grid1, 1024>>>(d_c0sy)
10 poly_sub_mod<<<grid1, 1024>>>(dev_dy, d_c0sy, dev_dy)
11 Copy the $dev\_dy$ from GPU global memory to CPU memory ($d\_y$).
12 round_extract_gmp(dy, d_y);
13 for i=0 ; i<batch_sz ; i++ do
14     round_extract_gmp(dy[i], $d\_y + i \times N_{mod} \times n$);
```

this technique effectively increases the parallelism compared to Algorithm 4.

6.3 Further Optimization: Merging Multiple Kernels

The input data are stored on the shared memory/registers to allow faster computation. However, these values are not kept when the kernel function is terminated, so the input data has to be reloaded from global memory to shared memory/registers repeatedly, whenever a new kernel is called. For example, referring to Algorithm 5, *dev_dy* used in line 6 is re-used in line 11, while *d_sk* are used in lines 7 and 9. To avoid unnecessary data transfers between different types of memories, we can merge all these kernels into one. Since all the execution takes place within a single kernel, we can utilize shared memory/registers without repeatedly copying the input to and from global memory. For instance, *dev_dy* used in line 6 can be stored in shared memory and reused to compute the polynomial subtraction in line 11. Once all execution completes, we can place the results on global memory. This simplifies the memory flow among different kernels and improves the performance. In our IPFE implementation, we always set the number of threads per block to 1024 for most of the functions, so that we can maximize the parallelism. This allows us to easily merge kernels that implement different functions, e.g., CRT conversion and multiplication in lines 5 and 6. However, recall that the extended three-level-combination proposed to speed up NTT computation only uses 512 threads. This poses a challenge when we merge the NTT and INTT kernels with the other kernels, due to the difference in number of threads per block. On one hand, if we use the maximum number of threads (1024) to compute NTT, half of the thread block is idle. On the other hand, if we reduce the block size in other kernels to match the one in NTT, we are sacrificing too much parallelism, which can degrade the overall performance. For this reason, we use the two-level-combination (instead of three-level) when merging all the kernels, since this technique uses 1024 threads.

Table 5: Execution times of IPFE SVM to compare the kernel merge technique on RTX 2060 (medium security level)

Functions	SVM with reference IPFE (3-level)	SVM with optimized IPFE (3-level)	SVM with merged IPFE (2-level)
Setup	35.25 ms	36.69 ms	36.82 ms
Encrypt	26.69 ms	**20.20 ms**	20.25 ms
KeyGen	4.68 ms	0.11 ms	**0.10 ms**
Decrypt	6.14 ms	0.97 ms	**0.95 ms**
Classification (100 data, 591 SVs)	6462.61 ms	701.51 ms	**687.96 ms**

Table 6: Comparing IPFE (medium security level) on CPU (AVX2 optimized implementation) and GPU implementation

functions	CPU (AVX2)	GPU (merged IPFE)	Speed-up AVX2 / GPU
Setup	1261.15 ms	**36.82 ms**	34.24×
Encrypt	810.66 ms	**20.25 ms**	40.02×
KeyGen	17.13 ms	**0.11 ms**	156.30×
Decrypt	17.97 ms	**0.96 ms**	18.76×
Classification (100 data, 591 SVs)	22786.21 ms	**687.97 ms**	33.12×

7 Experiment

In this section, we present and discuss the experimental results of the implementation techniques for SVM with IPFE on CPU and GPU. This experiment was conducted on a workstation with an Intel i7-9700F processor with eight cores, 16GB RAM, and an NVIDIA RTX 2060 (Turing architecture) GPU with 1920 cores and 6GB RAM. In the following experiments, we implement the RLWE-IPFE [MKMS22] scheme with medium security level, where $n = 4096$ and $l = 785$.

Table 5 shows the performance of SVM protected by the proposed cuFE, tested on a dataset from [CL11] with 100 data and 591 SVs. The optimized IPFE uses the three-level-combination technique to implement NTT, since it shows the best timing performance from our micro-benchmarking (see Section 5.3.2). In the case of `Setup`, the optimized IPFE does not enjoy any performance improvement over our reference GPU implementation. This is because `Setup` is only performed once and remain unchanged in the SVM classification process. In the case of `Encrypt`, the optimized IPFE is 1.32× faster than the reference, because it is processing 100 data in a batch. Similarly, the `KeyGen` and `Decrypt` are 42.5× and 6.32× faster than the reference, since 591 SVs are processed in a batch. To classify 100 data, the SVM applying optimized IPFE is $\approx 9\times$ faster than the reference version. Note that the reference version only relies on the fine-grain parallelism in IPFE, so the performance is less appealing in most cases. In contrast, the optimized IPFE can fully exploit the coarse-grain parallelism in SVM and the fine-grain parallelism in IPFE, eventually achieving faster performance compared to the reference version. In all cases excluding `Setup` and `Encrypt`, the merged IPFE is slightly faster than the optimized IPFE. To classify 100 data with 591 SVs, the merged IPFE took 688ms, which is roughly the same as the optimized version.

Table 6 compares the performance of RLWE-IPFE [MKMS22] protected SVM classification on CPU and GPU. The AVX2 implementation is provided by RLWE-IPFE [MKMS22] as a reference code and included in Table 6 for comparison. The proposed `Setup` implementation on GPU takes 36.82 ms, which is 34.24× faster than AVX2 implementation.

Performing Encrypt on a GPU takes 20.25 ms, which is 40.02× faster than the AVX2 implementation. KeyGen executed on a GPU enjoyed the most performance gain; it takes only 0.11 ms, which is 156.30× faster than AVX2 implementation. This is because cuFE processes 591 SVs at once to fully exploit the parallelism on a GPU. Note that KeyGen is used repeatedly for all the data and SVs, so the performance of this function is crucial in the practical use. Similarly, Decrypt function also enjoy 18.76× faster performance compared to AVX2 implementation, due to the huge parallelism available. However, the speed-up of Decrypt is not as good as KeyGen. This is because the inverse CRT and the final extraction of result involves large integer arithmetic, which is not trivially parallelizable on GPU. To classify 100 data with 591 SVs, our cuHE is 33.12× faster than AVX2 implementation.

8 Conclusion

The first implementation of IPFE on GPU devices, cuFE, was presented in this paper. We demonstrated that through several specially crafted implementation techniques, cuFE can achieve remarkable speed-up against the already optimized AVX2 implementation (from 18.76× to 156.30×) of a recently proposed RLWE-IPFE [MKMS22]. This opens up the possibilities to apply IPFE on various machine learning applications that are too complicated to be computed on the CPU. We anticipate that these results can facilitate the adoption of FE in many practical scenarios, as the overheads of FE schemes are often too costly for real applications. The proposed cuFE can be adopted easily by a cloud server or personal computing device that is equipped with a GPU. Note that a GPU-based solution is more flexible compared to other hardware accelerator like FPGA, as GPUs are commonly found in many cloud servers, workstations and laptops.

Acknowledgements

This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, the Research Council KU Leuven (C16/15/058), the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and SRC grant 2909.001.

Further, Angshuman Karmakar is funded by FWO (Research Foundation – Flanders) as junior post-doctoral fellow (203056 / 1241722N LV).

References

[ABCP15] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 733–751. Springer, 2015.

[ABKW19] Michel Abdalla, Fabrice Benhamouda, Markulf Kohlweiss, and Hendrik Waldner. Decentralizing inner-product functional encryption. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 128–157. Springer, 2019.

[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.

[ALS16] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 333–362, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[AWS] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/. Accessed: 2022-04-10.

[BCNS15] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570, May 2015.

[BDK⁺18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BG12] Alex Biryukov and Johann Großschädl. Cryptanalysis of the full aes using gpu-like special-purpose hardware. *Fundamenta Informaticae*, 114(3-4):221–237, 2012.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BJMS21] Milad Bahadori, Kimmo Järvinen, Tilen Marc, and Miha Stopar. Speed reading in the dark: Accelerating functional encryption for quadratic functions with reprogrammable hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–27, 2021.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, 2011.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19. ACM, 1988.

[CL11] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. volume 2, pages 1–27. ACM transactions on intelligent systems and technology (TIST), May 2011.

[CSG⁺18] Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. Decentralized multi-client functional encryption for inner product. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 703–732. Springer, 2018.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013.

[DSvW21] Léo Ducas, Marc Stevens, and Wessel van Woerden. Advanced lattice sieving on gpus, with tensor cores. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 249–279. Springer, 2021.

[FCLJ05] Rong-En Fan, Pai-Hsuen Chen, Chih-Jen Lin, and Thorsten Joachims. Working set selection using second order information for training support vector machines. *Journal of machine learning research*, 6(12), 2005. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/ (Accessed: 2020-01-09).

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru, 2020. [Online; accessed 10-April-2022].

[FS19] Tim Fritzmann and Johanna Sepúlveda. Efficient and flexible low-power NTT for lattice-based cryptography. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 141–150. IEEE, 2019.

[GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload – a cache attack on the bliss lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 323–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564. ACM, 2013.

[GMP] GNU multi-precision arithmetic library. https://gmplib.org/. Accessed: 2022-04-05.

[GVW12] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 162–179. Springer, 2012.

[HKR+18] James Howe, Ayesha Khalid, Ciara Rafferty, Francesco Regazzoni, and Máire O'Neill. On practical discrete gaussian samplers for lattice-based cryptography. *IEEE Trans. Computers*, 67(3):322–334, 2018.

[HPRR20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: From inception to implementation. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, volume 12100 of *Lecture Notes in Computer Science*, pages 53–71. Springer, 2020.

[IBM] IBM GPU cloud server. https://www.ibm.com/cloud/gpu. Accessed: 2022-04-10.

[JKA+21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 114–148, 2021.

[KLL21] Liang Kong, Shuguo Li, and Ruirui Liu. High-performance constant-time discrete gaussian sampling. *IEEE Trans. Computers*, 70(7):1019–1033, 2021.

[KRR+18] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Constant-time discrete gaussian sampling. *IEEE Trans. Computers*, 67(11):1561–1571, 2018.

[KRVV19] Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Pushing the speed limit of constant-time discrete gaussian sampling. a case study on falcon. Cryptology ePrint Archive, Report 2019/267, 2019. https://ia.cr/2019/267.

[LCFS17] Damien Ligier, Sergiu Carpov, Caroline Fontaine, and Renaud Sirdey. Privacy preserving data classification using inner-product functional encryption. In *ICISSP*, pages 423–430, 2017.

[LCPG16] Wai-Kong Lee, Hon-Sang Cheong, Raphael C-W Phan, and Bok-Min Goi. Fast implementation of block ciphers and prngs in maxwell gpu architecture. *Cluster Computing*, 19(1):335–347, 2016.

[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.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proceedings*, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[LS19] Ting Li and Yao Sun. Preimage attacks on round-reduced keccak-224/256 via an allocating approach. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 556–584. Springer, 2019.

[MKMS22] Jose Maria Bermudo Mera, Angshuman Karmakar, Tilen Marc, and Azam Soleimanian. Efficient lattice-based inner-product functional encryption. *The International Conference on Practice and Theory of Public-Key Cryptography (PKC)*, 2022.

[NVI22] CUDA NVIDIA. CUDA C programming guide, version 11.6. *NVIDIA Corp*, 2022.

[ÖEM+22] Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Efficient number theoretic transform implementation on gpu for homomorphic encryption. *The Journal of Supercomputing*, 78(2):2840–2872, 2022.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556, 2010. https://ia.cr/2010/556.

[ope] Opencl: Open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. Accessed: 2022-04-15.

[Pes16] Peter Pessl. Analyzing the shuffling side-channel countermeasure for lattice-based signatures. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, pages 153–170. Springer International Publishing, Cham, 2016.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *International conference on cryptology and information security in Latin America*, pages 346–365. Springer, 2015.

[Pol71] J. M. Pollard. The fast fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.

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

[RPB+19] Théo Ryffel, David Pointcheval, Francis Bach, Edouard Dufour-Sans, and Romain Gay. Partially encrypted deep learning using functional encryption. *Advances in Neural Information Processing Systems*, 32, 2019.

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

[YFX+21] Lihua Yin, Jiyuan Feng, Hao Xun, Zhe Sun, and Xiaochun Cheng. A privacy-preserving federated learning for multiparty data sharing in social iots. *IEEE Transactions on Network Science and Engineering*, 8(3):2706–2718, 2021.

[YSQW20] Haining Yang, Ye Su, Jing Qin, and Huaxiong Wang. Privacy-preserving outsourced inner product computation on encrypted database. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[ZSS19] Raymond K Zhao, Ron Steinfeld, and Amin Sakzad. FACCT: fast, compact, and constant-time discrete gaussian sampler over integers. *IEEE Transactions on Computers*, 69(1):126–137, 2019.

[ZYC+20] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 49–72, 2020.