IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 314–339. DOI:10.46586/tches.v2025.i1.314-339

GPU Acceleration for FHEW/TFHE Bootstrapping

Yu Xiao[1,2], Feng-Hao Liu[3], Yu-Te Ku[1,2,5], Ming-Chien Ho[1,2], Chih-Fan Hsu[1], Ming-Ching Chang[1,4], Shih-Hao Hung[2,6] and Wei-Chao Chen[1]

[1] Inventec Corporation, Taipei, Taiwan, {hsu.chih-fan,chen.wei-chao}@inventec.com
[2] National Taiwan University, Taipei, Taiwan, {r11922138,d08946006,r11944009,hungsh}@csie.ntu.edu.tw
[3] Washington State University, Pullman, USA, feng-hao.liu@wsu.edu
[4] State University of New York at Albany, Albany, USA, mchang2@albany.edu
[5] Academia Sinica, Taipei, Taiwan
[6] Mohamed bin Zayed University of Artificial Intelligence, Masdar, Abu Dhabi

Abstract. Fully Homomorphic Encryption (FHE) allows computations to be performed directly on encrypted data without decryption. Despite its great theoretical potential, the computational overhead remains a major obstacle for practical applications. To address this challenge, hardware acceleration has emerged as a promising approach, aiming to achieve real-time computation across a wider range of scenarios. In line with this, our research focuses on designing and implementing a Graphic Processing Unit (GPU)-based accelerator for the third generation FHEW/TFHE bootstrapping scheme, which features smaller parameters and bootstrapping keys particularly suitable for GPU architectures compared to the other generations.

In summary, our accelerator offers improved efficiency, scalability, and flexibility for extensions, e.g., functional bootstrapping (Liu et al., Asiacrypt 2022), compared to current state-of-the-art solutions. We evaluate our implementation and demonstrate substantial speedup in the single-GPU setting, our bootstrapping achieves an 18× - 20× speedup compared to a 64-thread server-class CPU; by using 8 GPUs, the throughput can be further improved by 7× compared to the single-GPU implementation, confirming the scalability of our design. Furthermore, compared to the SoTA GPU solution TFHE-rs, we achieve a maximum speedup of 1.69× in AND gate evaluation. Finally, we benchmark several private machine learning applications, showing real-time solutions for (1) encrypted neural network inference for MNIST in 0.04 seconds per image, which is the fastest implementation to our knowledge.(2) private decision trees in 0.38 seconds for Iris dataset, where as prior 16 cores CPU implementation (Lu et al., IEEE S&P 2021) required 1.87 seconds; These results highlight the effectiveness and efficiency of our GPU-acceleration in real-world applications.

As a technical highlight, we design a novel parallelization strategy tailored for FHEW/TFHE bootstrapping, allowing an automated optimization that partitions bootstrapping into multiple GPU thread blocks. This is necessary for FHEW/TFHE bootstrapping with scalable parameters, where the whole bootstrapping process may not fit into a single thread block. With this, our accelerator can support a broader range of parameters, making it ideal for upcoming privacy-preserving applications.

Keywords: Fully Homomorphic Encryption · Bootstrapping · GPU Acceleration

1 Introduction

Fully Homomorphic Encryption (FHE) is a transformative technology that enables computations on encrypted data without decryption. Since Gentry's initial and seminal

Licensed under Creative Commons License CC-BY 4.0.
Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

work [Gen09], there has been extensive subsequent work, e.g., [Bra12, BGV12, FV12, DM15, CGGI16, CKKS17], improving the design and efficiency in both theory and practice. Despite the efforts, there are still substantial obstacles in many applications due to the computational overhead. For example, the work [KS23] performed an encrypted neural-network image inference over the simple MNIST dataset, yet the computation time can take around an hour (per image) even using Intel HEXL [BKS+21] library. Thus, accelerating FHE solutions remains a crucial and compelling research direction.

Hardware acceleration, including options like GPU, FPGA, and ASIC, has emerged as a promising avenue for practical FHE-based solutions [ZCY+22]. Given that current FHE computations primarily involve polynomial operations of high degrees, effective hardware parallelization can significantly reduce computation time. Furthermore, in many machine learning applications that are inherently parallelizable—such as image classification, where computations involve simple operations but over extensive datasets—the impact of hardware acceleration would become even more pronounced. Following this line of research, this work aims to develop a CUDA implementation for FHE computation on GPUs, significantly extending its practical applicability to broader domains.

Focus of this work. Among various approaches, we particularly focus on GPU-based acceleration with FHEW/TFHE [DM15, CGGI16], known as the third generation of FHE designs. This approach leverages the architectural advantages of GPUs and the fast-bootstrapping feature of FHEW/TFHE, as outlined below:

- GPUs consist of a lot of small computing units, e.g., streaming multiprocessors (SM), which naturally accelerate parallelizable computations. While each unit may not be as powerful (in terms of speed and memory size) as high-end CPUs, the large number of parallel units can significantly reduce overall computation time with appropriate designs for major applications.
- FHEW/TFHE possess the feature of small FHE parameters, resulting in substantially smaller bootstrapping keys (10 to 100 MB) compared to other generations of schemes like CKKS [CHK+18] (1 to 10 GB). This simplifies the design by enabling the allocation of smaller FHE objects into the GPU computing units.

Combining the advantages, our goal is to design and implement an FHE accelerator with improved efficiency, scalability, and flexibility for extensions over prior works and the state of the art [Zam22].

Bottleneck of FHE computation. Bootstrapping [Gen09, Gen10], used to refresh the accumulated error in ciphertext, stands as the bottleneck operation in all current designs of FHE. The fundamental concept of bootstrapping is to homomorphically decrypt the ciphertext, allowing the removal of accumulated errors during decryption, which involves large amount of computations. In our approach, we leverage the immense computational power of GPUs to concurrently execute multiple bootstrappings, thereby enhancing the overall throughput of bootstrapping operations. Broadly, FHEW/TFHE bootstrapping falls into two categories: regular and functional. Regular bootstrapping serves the primary function of error refreshing in ciphertext. Gate evaluation using regular bootstrapping to evaluate logical gateways such as AND and OR gates is an example. Functional bootstrapping retains the core functionality of bootstrapping while simultaneously enabling arbitrary look-up table evaluations, enriching the FHEW/TFHE to evalaute non-polynomial functions over the plaintext encoding simultaneously. However, standard functional bootstrapping incurs a high computational cost to enhance numerical precision (requiring larger FHE parameters, e.g., plaintext, modulus, and ring dimension). Large-precision functional bootstrapping [LMP22] addresses this challenge by performing multiple smaller bootstrappings instead of a single large bootstrapping, effectively enhancing precision without drastically increasing execution time.

In this work, we focus on accelerating the bootstrapping operations in FHEW/TFHE by harnessing the power of graphic processing units (GPUs). Mainly, we introduce a parallelization strategy that expands our GPU-based bootstrapping solution's range of supported parameters, realizing bootstrapping from regular to large-precision variants. By several rigorous optimizations, our proposed solution exhibits superior performance, paving the way for more efficient and practical FHE in broader application domains.

1.1 Prior Works

There are two major earlier works of GPU-accelerated FHEW/TFHE bootstrapping implementations, cuFHE [ver18] and NuFHE [nuc18]. Both implementations adhere to the original TFHE bootstrapping specification of Chillotti et al. [CGGI16], and are designed to execute the gate bootstrapping exclusively. However, both cuFHE and NuFHE encounter three primary challenges in their implementations.

Firstly, both GPU implementations adopt fixed cryptographic parameters (somewhat hardcoded in the implementation and optimization), adhering to an 80-bit security level for gate bootstrapping. As noted in [ZCY+22], these cryptographic parameters are fixed and cannot be reconfigured for greater generality. Specifically, parameters such as ring dimension or ring modulus cannot be altered by simply changing values in parameter tables; one needs to trace the entire GPU source code and modify the desired parameters wherever they appear. Their designs do not consider flexible parameters, making it challenging to adopt 128-bit level parameters.

> **Challenge 1**. The fixed/hardcoded parameters make it difficult to adjust the HE parameters, while different privacy-preserving applications may require different parameter configurations.

Additionally, even if one modifies the desired parameters throughout the source code, the supported parameter range is limited due to the parallelization strategy employed in both GPU implementations. These works utilize aggressive optimization methods aimed at fitting a bootstrapping operation into a single Streaming Multiprocessor (SM) of the GPU. However, as parameters increase, this approach can easily exceed the limitations of an SM, such as number of threads or amount of shared memory.

> **Challenge 2**. The parallelization strategy used in these works has a low upper bound in supported parameters, thus not supporting scalable parameters required for higher security levels or (functional) bootstrapping.

Lastly, extending these implementations to support large-precision (functional) bootstrapping introduces complexities. These extensions necessitate a more scalable parameter set compared to gate bootstrapping and require frequent parameter adjustments for various functions.

> **Challenge 3**. Extending to functional bootstrapping or its large-precision variants [LMP22] incurs challenging due to the constraints posed by both fixed parameters and the parallelization strategy.

Other related works. For other FHE schemes like BGV/BFV [Bra12, BGV12, FV12] and CKKS [CKKS17], there are two notable GPU-accelerated works: [JKA+21] and TensorFHE [FWX+23]. [JKA+21] achieved a 40.0× speedup compared to previous 8-thread CPU implementations by running a logistic regression model training. As a successor, TensorFHE further improved the multiplication and rotation operations in CKKS, achieving 1.35× and 1.41× speedup over [JKA+21], respectively. Both works demonstrate the significant potential of using GPUs to accelerate operations in Fully Homomorphic Encryption. Since the parameters in BGV/BFV and CKKS are typically much larger than those in the FHEW/TFHE scheme, the challenges these works face in

accelerating using GPUs are very different from those encountered in FHEW/TFHE. In this work, we specifically focus on identifying the bottlenecks in expediting the FHEW/TFHE scheme using GPUs.

1.2 Major Contributions

We present a GPU-accelerated FHEW/TFHE bootstrapping implementation that addresses the three challenges mentioned above. More concretely, our contributions can be summarized as follows:

- We propose a novel parallelization strategy for GPU-accelerated FHEW/TFHE bootstrapping. This strategy adaptively partitions the bootstrapping workload based on the FHE parameters and balances these workloads across multiple GPU thread blocks. Using this strategy, we can configure a wide range of FHE parameters to meet a wide range of application requirements without exceeding GPU thread block constraints, such as the number of threads per thread block.
- We conduct a comprehensive numerical and empirical error analysis to assess the additional error introduced by the precision loss in FFT-based polynomial multiplication implementation, which is an important basic operation in FHEW/TFHE bootstrapping. This includes revising the previous error analysis metrics for bootstrapping by analyzing the precision loss stemming from floating-point operations and evaluating the errors of parameters at various security levels as in OpenFHE [BBB+22] and all functional bootstrapping parameters introduced in large-precision functions [LMP22]. While FFT provides excellent flexibility, it is important to note that this paper does not fully explore the potential of NTTs with sparse primes.
- We integrate the above technical contributions into the OpenFHE lattice cryptography library [BBB+22], introducing several implementation-specific optimizations such as the fused multiply-add operation during polynomial multiplications. Our designs naturally support the large-precision functional bootstrapping functions [LMP22] in the OpenFHE library and are the first GPU implementation realizing these functions, including EvalFunc, EvalFloor, EvalSign, and EvalDecomp.
- We evaluate our GPU implementation across various applications to validate its effectiveness and efficiency. Our approach improves the previous state-of-the-art in speed for non-linear functions, encrypted neural network inference, and private decision tree evaluations, as detailed in Table 1. Our experiments demonstrate the immense potential of real-time FHE applications using GPU acceleration.

For details, we compared our GPU implementations with a 64-thread CPU implementation on gate bootstrapping and large-precision functional bootstrapping. Our single-GPU implementation demonstrated a substantial performance boost compared to the 64-thread CPU implementation, e.g., 18× - 20× speedup. Our 8-GPU implementation boosts nearly 7× compared to the single-GPU version, showing the scalability of our design. We compare our method with the most recent GPU implementation on the FHEW/TFHE scheme, TFHE-rs [Zam22], with the proposed benchmarks of TFHE-rs on the AND gate. The results demonstrated our superiority in most scenarios, with a maximum speedup of 1.69×.

We have applied methods to three applications: (1) general non-linear functions, (2) neural network inference, and (3) decision trees. We briefly compare our method with the SoTAs on three different applications in Table 1. Specifically, for the general non-linear functions, such as ReLU and max-pooling, which are commonly used in neural networks, our implementation boosts 30× in end-to-end runtime compared with [jLHH+21]. For encrypted neural network inference on the MNIST dataset, the average processing time is 0.04 second per image with 128 bits security level, while prior best known solution [BMMP18] required 0.14 seconds but with lower security level (80 bits). For a private decision tree,

Table 1: Comparison of the three applications between our work and the previous SoTAs. [jLHH+21] used a 20-thread CPU in non-polynomial functions, and a 16-thread CPU in Decision tree. [LLZ+24] used a 16-core CPU.

Application	Ours	Previous SoTA	Speedup
Non-polynomial Functions	1.59 ms	48.14 ms [jLHH+21]	30.3×
Neural Network Inference	0.04 s	0.14 s [LLZ+24]	3.5×
Decision Tree	0.38 s	1.87 s [jLHH+21]	4.9×

our 1-GPU implementation achieves real-time evaluation in 0.38 seconds, outperforming the previous approach [jLHH+21] that took 1.87 seconds. These applications demonstrate the practicality of our GPU implementation and underscore the immense potential of FHE-based privacy-preserving solutions.

2 Preliminaries

In this section, we delve into the fundamental operations employed in the bootstrapping process of FHEW/TFHE [DM15, CGGI16], as utilized within the OpenFHE framework [BBB+22], detailed in Section 2.1. In Section 2.2, we comprehensively compare the NTT-based bootstrapping implementation with the FFT-based one, aiming to elucidate their distinctions and functionalities. In Section 2.3, we explore the limitation of shared memory and the number of threads in GPU, which has previously constrained the flexibility of bootstrapping running on GPU.

2.1 Bootstrapping in FHEW/TFHE Cryptographic System

LWE encryption scheme. Learning with Errors (LWE) is the fundamental cryptography in FHEW/TFHE, offering a robust framework for constructing secure cryptographic systems. The core of LWE involves solving a computational problem where one must distinguish between random noise and structured data in noisy linear equations. An LWE ciphertext takes the form of $(\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$. There are two prominent LWE variants: Ring LWE (RLWE) and Ring Gentry-Sahai-Waters (RGSW). RLWE operates within polynomial rings, leveraging the properties of polynomial rings to create more efficient cryptographic constructions. RGSW builds upon RLWE, exceling in noise control during operations and serves as a crucial cornerstone for subsequent FHEW/TFHE developments.

We first introduce the notations relevant to the construction of RLWE and RGSW.

- n : the polynomial dimension of LWE scheme.
- q : the modulus of LWE scheme.
- N : the polynomial dimension of RLWE scheme.
- Q : the modulus of RLWE scheme.
- B_g : the base of RGSW scheme for gadget decomposition.
- d_g : the gadget decomposition length of RGSW scheme.

RLWE and RGSW encryption scheme. Let $R = \mathbb{Z}[X]/(X^N + 1)$, for N is a power of 2. $R_Q = R/QR \equiv \mathbb{Z}_Q[X]/(X^N + 1)$. RLWE encrypts a polynomial $\widetilde{m} \in R_Q$ under secret key $s \in R$ as

$$\mathsf{RLWE}_s(\widetilde{m}) = (a, as + e + \widetilde{m}),$$

where $a \in R_Q$ is sampled from random, and $e \in R$ where its coefficients are sampled from discrete zero-mean Gaussian distribution φ_σ. To mitigate noise growth during scalar multiplication with RLWE, RLWE′ is introduced. The encryption of a polynomial $\widetilde{m} \in R_Q$

under the secret key $s \in R$ using RLWE' is defined as

$$\mathsf{RLWE}'_s(\widetilde{m}) = (\mathsf{RLWE}_s(\widetilde{m}), \mathsf{RLWE}_s(B_g\widetilde{m}), \mathsf{RLWE}_s(B_g^2\widetilde{m}), ..., \mathsf{RLWE}_s(B_g^{d_g-1}\widetilde{m})),$$

where the base B_g balances the running time and the noise growth rate when performing the encrypted operations, and $d_g = \log_{B_g} Q$. The fundamental concept behind RLWE' is to control the upper bound of error growth during scalar multiplication with RLWE, ensuring that the scalar will not directly amplify the error. To facilitate multiplication between ciphertexts, the ring variant of the third-generation FHE scheme GSW [GSW13], known as RGSW, is used. Built upon RLWE' using the same secret key, RGSW is defined as

$$\mathsf{RGSW}_s(\widetilde{m}) = (\mathsf{RLWE}'_s(-s \cdot \widetilde{m}), \mathsf{RLWE}'_s(\widetilde{m})).$$

Next, we introduce an operation called gadget decomposition (GDecomp) designed to decompose an RLWE ciphertext based on the chosen basis B_g. This operation is crucial for supporting multiplication between RLWE and RGSW. Its definition is as follows:

$$\mathsf{GDecomp}(\mathsf{RLWE}(\widetilde{m}) = (a_0, b_0, \ldots, a_{d_g-1}, b_{d_g-1}),$$

$$\text{where } \mathsf{RLWE}(\widetilde{m}) = (a, b), a = \sum_{i=0}^{d_g-1} a_i B_g^i, b = \sum_{i=0}^{d_g-1} b_i B_g^i.$$

Finally, the multiplication between RLWE and RGSW is referred to as the external product in [CGGI16]. This operation achieves good noise control during multiplication between ciphertexts, which is defined as

$$\mathsf{GDecomp}(\mathsf{RLWE}(m_0)) \diamond \mathsf{RGSW}(m_1) = \mathsf{RLWE}(m_0 \cdot m_1).$$

For detailed information on these encryption schemes, refer to [MP21]. In the following, we delve into the bootstrapping operation in the FHEW/TFHE cryptographic system.

FHEW/TFHE bootstrapping. The general paradigm of FHE bootstrapping is to perform its decryption function homomorphically so that the accumulated error the ciphertext can be refreshed. This method was invented by Gentry [Gen09] and then became the foundation of all currently in-use FHE schemes.

For the focus of this work – the third generation FHEW/TFHE, the bootstrapping procedure takes inputs an LWE ciphertext $(\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, an encrypted secret key vector $\mathbf{s} \in \mathbb{Z}_q^n$, and outputs a refreshed LWE ciphertext encrypting the same message. The particular decryption function of an LWE ciphertext can be expressed as $\lfloor b - \langle \mathbf{a}, \mathbf{s} \rangle \bmod q \rceil$, where $\lfloor \cdot \rceil$ is some rounding function. This decryption procedure can be computed by an NC1 circuit (with some dimension and modulus reduction techniques [BV11]), and the work [BV14] further showed that a polynomial modulus suffices for the homomorphic computation by using the asymmetric noise growth of the GSW [GSW13] and converting the NC1 decryption circuit to a polynomial-length branching program. However, this design principle is for theoretical feasibility, and it was not clear how to realize a concretely efficient bootstrapping method.

Shortly, the work [AP14] proposed the first explicit bootstrapping method within a polynomial modulus under LWE, eliminating the need for conversion between NC1 and branching programs. Following this, FHEW [DM15] and TFHE [CGGI16] significantly enhanced the concrete efficiency by leveraging the ring structure and external products. These advancements in practical bootstrapping brought the operation time down to within 100 ms. In a nutshell, the FHEW/TFHE method aligns the input LWE modulus q with $2N$ where N is the ring dimension via the modulus switch technique. Then once can homomorphically compute $z = b - \langle \vec{a}, \vec{s} \rangle$ over the exponent, resulting in a ciphertext of $\mathsf{RLWE}(X^z)$. This step is known as the Blind Rotate (according to [CGGI16]) or the

Update procedure as Algorithm 1. From here, the FHEW/TFHE uses a clever extraction technique that outputs a ciphertext of $\mathsf{LWE}(\lfloor z \rceil)$, resulting in the desired outcome.

We summarize the FHEW/TFHE precedure below:

- **Initialization**: This phase involves encrypting the $\mathsf{LWE}^{n/q}$ ciphertext into an $\mathsf{RLWE}^{N/Q}$ (cryptographic accumulator) without introducing additional noise.
- **Update**: This is the bottleneck phase where the accumulator is updated using a bootstrapping key, performing the homomorphic decryption of the ciphertext.
- **Extract**: In this phase, the output $\mathsf{RLWE}^{N/Q}$ from the update phase is extracted back into a $\mathsf{LWE}^{N/Q}$ ciphertext.
- **Keyswitch**: The $\mathsf{LWE}^{N/Q}$ ciphertext undergo a dimension reduction process using a keyswitching key to reduce the dimension N back to n, i.e. $\mathsf{LWE}^{N/Q} \to \mathsf{LWE}^{n/Q}$.
- **Modswitch**: Finally, we do a modulus reduction on the ciphertext $\mathsf{LWE}^{n/Q}$ to bring the modulus Q down to q. The output is a refreshed ciphertext $\mathsf{LWE}^{n/q}$.

For the Blind Rotate/Update, we adopt the enhanced GINX method proposed by [BIP+22], also called TFHE/GINX by [BBB+22]. We aim to accelerate the update procedure (ref. Algorithm 1) as this phase is the most time-consuming process in the whole procedure. Notably, the operation in Line 4 involves a large number of polynomial multiplications, i.e., $n \times (8 \times d_g + 4)$ polynomial multiplications, where $8 \times d_g$ is for the $\mathsf{ACC} \diamond \mathsf{RGSW}$ operation, and 4 is for the $(x^{a_i} - 1)(\mathsf{ACC}_{\mathsf{decomp}} \diamond \mathsf{RGSW}_{0i})$ and $(x^{-a_i} - 1)(\mathsf{ACC}_{\mathsf{decomp}} \diamond \mathsf{RGSW}_{1i})$ operations, and d_g is the gadget decomposition length of RGSW.

Algorithm 1: Update

input : A RLWE ciphertext ct_0;
a vector $\mathbf{a} = (a_1, ..., a_n) \in \mathbb{Z}_q^n$;
the bootstrapping key $\{\mathsf{RGSW}_{0i}, \mathsf{RGSW}_{1i}\}_{i \in [1,n]}$;
output : A refreshed RLWE ciphertext

1. $\mathsf{ACC}_0 \leftarrow \mathsf{ct}_0$
2. **for** $i \leftarrow 1$ **to** n **do**
3. $\quad \mathsf{ACC}_{\mathsf{decomp}} \leftarrow \mathsf{GDecomp}(\mathsf{ACC}_{i-1})$
4. $\quad \mathsf{TEMP} \leftarrow (x^{a_i} - 1)(\mathsf{ACC}_{\mathsf{decomp}} \diamond \mathsf{RGSW}_{0i}) + (x^{-a_i} - 1)(\mathsf{ACC}_{\mathsf{decomp}} \diamond \mathsf{RGSW}_{1i})$
5. $\quad \mathsf{ACC}_i \leftarrow \mathsf{ACC}_{i-1} + \mathsf{TEMP}$
6. return ACC_n

2.2 NTT- and FFT-based Bootstrapping Implementations

Leveraging number theoretic transform (NTT) or fast Fourier transform (FFT) to accelerate the polynomial multiplication is widely used to reduce the execution time of Bootstrapping. Asymptotically, FFT/NTT can reduce the time complexity of polynomial multiplication from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. In Figure 1, we illustrate the flow chart for using FFT to accelerate polynomial multiplications during the update phase of bootstrapping of Algorithm 1. Before the polynomial multiplications with the bootstrapping key and monomials, the polynomials in the decomposed accumulator is transformed in to the FFT form. After the polynomial multiplications, the polynomials in the output accumulator apply IFFT to return to the coefficient representation.

NTT and FFT have their pros and cons. Generally, NTT generates the *exact* output at a slower speed, while FFT produces an *approximate* output faster. Notably, some implementations of NTT, such as those using the Goldilocks prime $Q = 2^{64} - 2^{32} + 1$, can leverage a shift-add method for modular reduction, making it potentially faster than FFT. However, the parameter settings for FHE can be broad, and it may not always be possible

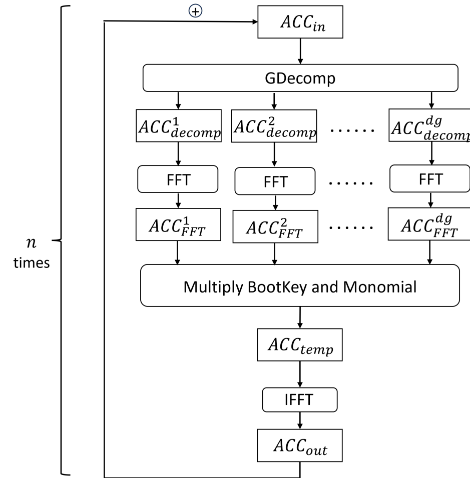


Figure 1: Flow chart of the update phase in FHEW/TFHE bootstrapping with polynomial multiplication accelerated by FFT.

to select sparse primes in the targeted scenario. Thus in our work, we opted for FFT because it imposes no constraints on the modulus, offering greater flexibility and efficiency in general. For other works, the decision to use FFT or NTT depends on the specific use case and requirements. For instance, in between OpenFHE [BBB+22] and TFHE-rs [Zam22], OpenFHE adopts NTT for polynomial multiplication, while TFHE-rs opts for FFT in its implementation, respectively. NuFHE [nuc18] has both FFT-based and NTT-based bootstrapping on GPUs, and Zhang et al. [ZCY+22] compared the relative speed between the two bootstrapping implementations. They have shown that the FFT-based bootstrapping in NuFHE [nuc18] is 2.3× faster than their NTT-based one.

For the parameter setting of third-generation FHE, i.e., $N = 2^{10} \sim 2^{12}$ and $Q \leq 2^{64}$, we observe an efficiency advantage for FFT-based over NTT-based implementation in our experiments, even though these two methods have the same asymptotic efficiency. We identify two critical factors: (1) NTT often suffers from higher constant factors due to the additional steps required for modulus reduction when multiplying integers in a finite field. For instance, FFT's pointwise multiplication of Gaussian numbers follows the formula $(a + bi) \times (c + di) = (ac - bd) + (ad + bc)i$, which involves four floating-point multiplications and two additions. In contrast, integer multiplication in NTT may necessitate fast modulus reduction techniques, such as Barrett reduction, which introduces extra integer multiplications, shifts, and conditional operations, increasing computational overhead. (2) Additionally, FFT benefits from an extensive ecosystem of highly optimized tools and libraries, such as NVIDIA's cuFFT [NVI23] for GPU and FFTW [FJ98] for CPU, which are both fast and easy to integrate. In contrast, we are not aware of similarly optimized libraries for NTT.

We observe that simply replacing the NTT operation with FFT in the bootstrapping process of the OpenFHE library results in significant speedup. According to benchmarking results on the TFHE-rs website [Zam22], they compare the bootstrapping performance of the OpenFHE library, which uses NTT, and their library, which uses FFT, under the same 128-bit security level with modulus size under INT64 and the same machine. The OpenFHE implementation achieves a bootstrapping runtime of 24 ms, whereas TFHE-rs achieves 13.5 ms. While slightly different FHE parameters are used, the primary reason for the speedup is the highly optimized FFT library (SPQLIOS-FFT) used in TFHE-rs. Furthermore, GPUs are inherently designed for floating-point arithmetic, which is central to FFT operations, making FFT the preferred choice for our case.

2.3 GPU Shared Memory and the Number of Threads

Optimizing parallel processing tasks within the constraints of GPU resources is pivotal. In the context of NVIDIA Compute Unified Device Architecture (CUDA) programming, a thread block serves as the fundamental unit of execution. Each thread block is bound by a maximum limit on the number of threads, determined by factors such as the GPU model and its computing capability. Additionally, shared memory, a rapid yet restricted-access memory space shared among threads within a block, introduces another constraint. The amount of shared memory available per thread block varies across GPUs and significantly impacts the efficiency of data sharing and communication among threads. Striking a balance among these constraints is essential for achieving optimal performance in parallel applications. FHEW/TFHE Bootstrapping demands relatively low computational effort compared to other FHE schemes, such as CKKS [CKKS17]. In light of this, prior GPU works [ver18, nuc18] executed bootstrapping in a single thread block of the GPU.

In CUDA, the maximum thread count per thread block is either 512 for Compute Capability 1.x (pre-Fermi) or 1024 for newer compute capabilities. In previous GPU works, cuFHE [ver18] allocating 512 threads per block and NuFHE [nuc18] allocating 256 threads per block. However, the number of threads needed per block is influenced by the adjusted parameter set, particularly the increased value of d_g for the same security level. This adjustment poses a challenge as it brings these GPU works closer to the limitations of GPU resource constraints.

The maximum shared memory per thread block varies significantly across different compute capabilities. Recent GPUs' shared memory amounts varies from 96KB (V100) to 227KB (H100). Both cuFHE [ver18] and NuFHE [nuc18] aim to leverage shared memory to store decomposed ciphertext, i.e., $\mathsf{GDecomp}(\mathsf{RLWE})$, to minimize global memory fetching and storing. For instance, cuFHE [ver18] allocates 48KB of shared memory per bootstrapping. However, this shared memory usage pattern restricts the scalability of parameter sets in both related works, making them difficult to support bootstrapping with large parameters through simple modifications.

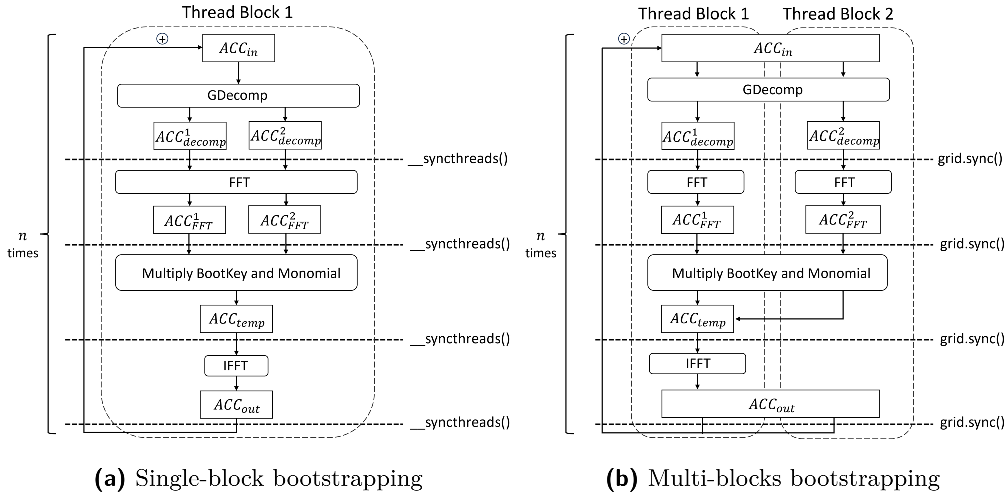
3 Methodology

We first point out the difficulties of supporting a wider range of parameters on GPU using previous parallelizing techniques [ver18, nuc18]. Both previous works run FHEW/TFHE bootstrapping within a single GPU thread block, highlighting temporal and spatial locality advantages. However, as parameter sets expand, the number of threads and shared memory usage can easily exceed GPU limitations. In Section 3.1, we introduce our novel parallelizing strategy called multi-blocks bootstrapping, offering a scalable approach that adjusts the number of thread blocks based on the decomposition length (d_g). This adaptive methodology aims to circumvent excessive resource utilization within a single thread block, accommodating a broader range of parameter sets.

Additionally, previous work NuFHE [nuc18] primarily focuses on using FFT in gate bootstrapping with small Q value (32 bits), where the precision loss incurred by FFT can be ignored. However, precision loss in floating-point arithmetic in FFT is inescapable when addressing functional bootstrapping, where the Q typically surpasses 32 bits. In Section 3.2, we revise the error analysis metric proposed by Micciancio et al. [MP21] with the additional FFT error. We conduct comprehensive experiments to verify the refined metric and re-estimate the decryption failure probabilities of existing parameter sets using FFT-based bootstrapping.

3.1 Multi-Blocks Bootstrapping

Previous GPU-accelerated FHEW/TFHE bootstrapping works [ver18, nuc18] run bootstrapping in a single GPU thread block, and we call it single-block bootstrapping in



(a) Single-block bootstrapping **(b)** Multi-blocks bootstrapping

Figure 2: Bootstrapping parallelizing strategies comparison between single-block and multi-blocks. We pick $d_g = 2$ as an example, while this strategy generalizes to arbitrary d_g values. ACC consists of two polynomial, `__syncthreads()` is used to synchronize threads in thread block in CUDA, and `grid.sync()` is used to synchronize threads in multiple thread blocks.

the following paragraphs. Single-block bootstrapping is feasible in scenarios involving smaller parameters, such as gate evaluation with a lower security level. However, when transitioning to functional bootstrapping, this strategy can easily surpass the limitations of GPU shared memory and number of threads per thread block. To address these limitations, we propose a novel parallelizing strategy called multi-blocks bootstrapping. Our approach dynamically adjust the GPU configuration based on the FHE parameters, enabling it to accommodate larger parameter sets and facilitate functional bootstrapping. It is important to note that we did not modify the bootstrapping algorithm but focused on the data arrangement on GPU during bootstrapping.

In Figure 2, we illustrate the parallelizing design of single-block bootstrapping and multi-blocks bootstrapping. Figures 3a and 3b both follow the update phase in the bootstrapping process outlined in Algorithm 1 and the flow chart in Figure 1. We employ FFT to accelerate the polynomial multiplications involved in bootstrapping. During bootstrapping, we observed that within the GDecomp operation in bootstrapping, an ACC, comprising two polynomials, is decomposed into d_g pairs of polynomials. Subsequently, these polynomials undergo FFT computation and polynomial multiplications. Single-block bootstrapping employs a single thread block to manage these polynomial computations, quickly encountering limitations discussed in Section 2.3 as parameters expand. Conversely, we launch d_g thread blocks with smaller sizes in multi-blocks bootstrapping, each responsible for handling the computations of one decomposed ACC. In this way, the workload is evenly distributed through all thread blocks. Each thread block is the same size as others and executes identical instructions but with different pairs of polynomials, aligning well with the GPU's execution model.

We illustrate our novelty by showing the kernel function configuration of multi-blocks bootstrapping in Listing 1. We set the number of thread blocks in the grid to be d_g, following the concept in Figures 3b. For the number of threads and the amount of shared memory in a thread block, since the FFT operation in bootstrapping requires the most GPU resources to be fully parallelized, these configurations follow the requirements in the cuFFTDx library [NVI23]. This library takes the FFT dimension and the specific GPU as inputs, determining the number threads and the amount of shared memory required. Lastly, the kernel function for bootstrapping is relatively small, so we use CUDA streams

to asynchronously send/receive data between the CPU and GPU and execute multiple bootstrappings on the GPU. This configuration allows us to achieve high throughput of bootstrapping, as we simultaneously execute tens to hundreds of bootstrappings on a GPU.

Furthermore, coordinating synchronization among these thread blocks introduces complexities. In Figures 3a, where only one thread block is utilized, using **`__syncthreads()`** in CUDA C++ achieves synchronization for all threads within the same thread block. However, for synchronizing multiple thread blocks, we invoke the bootstrapping kernel through **`cudaLaunchCooperativeKernel`** in the Cooperative Groups library (available since CUDA 9). Utilizing this API enables synchronization via **`grid.sync()`** to synchronize thread blocks within the grid, as long as the provided number of thread blocks does not exceed the maximum occupancy of the target GPU. Although the overhead of synchronization between thread blocks is usually high, in our case, since the number of thread blocks launched in the bootstrapping kernel function is d_g, and the value of d_g is usually small (2 to 10), the overhead of synchronizing these thread blocks is low. After extensive experiments, we measure only a 10% to 20% performance overhead due to synchronization. Thus, we take this approach in our design.

```
1 cudaLaunchCooperativeKernel(
2     (void*)(bootstrappingMultiBlock<FFT, IFFT>), // kernel function
3     dg, // number of thread blocks in a grid
4     FFT::block_dim, // number of threads in a thread block
5     kernelArgs, // arguments for kernel funcion
6     FFT::shared_memory_size, // shared memory size in a thread block
7     stream // CUDA streams
8 );
```

Listing 1: Launch multi-blocks bootstrapping kernel using CUDA cooperative groups API

Implementation-specific optimizations for bootstrapping We introduce two primary performance optimization techniques for our GPU bootstrapping implementation. Firstly, we apply the efficient negacyclic FFT algorithm proposed by Klemsa et al. [Kle21]. This FFT algorithm performs a folding step before the FFT to fold a polynomial of dimension N to $N/2$, and unfolds the polynomial back to N after IFFT. We observe that unfolding the polynomial back after IFFT is unnecessary; instead, we directly operate on the folded polynomials throughout the entire bootstrapping algorithm. In this way, the memory usage of these polynomials is halved, and the operations in the bootstrapping process are also reduced. Secondly, we use the CUDA extended instruction called the fused multiply-add (FMA) operation during the Multiply BootKey and Monomial phase in Figure 2. The operations involved in this phase are primarily multiplication and addition between polynomials. The FMA operation can execute the multiplication and addition of two floating-point numbers in one GPU instruction, which accelerates polynomial operations with the bootstrapping key and monomials. NVIDIA's whitepaper [WFF11] discusses the speed and precision enhancements achieved by using the FMA operation during floating-point computations. Compared to the unfused version, applying the FMA operation results in a 1.5× speedup during the Multiply BootKey and Monomial phase. Overall, this optimization provides a 1.1× to 1.2× speedup of the entire bootstrapping operation, depending on the parameter sets.

Executing bootstrapping using multi-blocks approach offers two significant advantages. Firstly, it supports a broader range of parameter sets, as illustrated in Table 2, since the increased workload associated with larger parameters is distributed to multiple thread blocks. Therefore, larger parameter sets from large-precision functional bootstrapping [LMP22] are naturally supported by using our approach. Moreover, we support the arbitrary value of d_g—a parameter balancing the error growth rate and running time during bootstrapping. Adjusting d_g becomes crucial if the application requires lower error, as it helps mitigate

error accumulation in bootstrapping. Secondly, the multi-block bootstrapping approach yields faster execution times when the number of bootstraps is smaller than the total number of Streaming Processors (SMs) across all GPUs. By distributing the workload across multiple thread blocks, each block may execute on different SMs simultaneously. Consequently, the GPU utilization rate increases as more SMs are engaged in processing, leading to improved overall performance.

Table 2: Comparison of supported parameter range for single-block bootstrapping and multi-blocks bootstrapping when $\mathsf{elem}_{\mathsf{FFT}} = 8$

Parameter	Single-block	Multi-blocks
N	512, 1024, 2048	512, 1024, 2048, 4096, 8192
d_g	$N = 512$: 1 to 16 $N = 1024$: 1 to 8 $N = 2048$: 1 to 4	arbitrary value

Detailed GPU resources usage calculation The determination of supported parameter sets shown in Table 2 is contingent upon calculating the number of threads and the amount of shared memory utilized. These GPU resource usages are intricately linked to FHE parameters and GPU settings. The number of threads in a bootstrapping kernel typically aligns with the requirements of the FFT. Let $\mathsf{elem}_{\mathsf{FFT}}$ denote the number of elements a thread manages in FFT. For single-block bootstrapping, a minimum of $N \times d_g \,/\, \mathsf{elem}_{\mathsf{FFT}}$ threads is necessary to parallelize each bootstrapping step fully. On the other hand, for multi-blocks bootstrapping, a minimum of $N / \mathsf{elem}_{\mathsf{FFT}}$ threads is required. Shared memory usage varies based on the implementation. In our multi-blocks bootstrapping approach, shared memory usage aligns with the minimum requirements stipulated by the cuFFTDx library [NVI23], which is $N \times \mathsf{elem}_{\mathsf{FFT}}$ bytes per thread block. This shared memory requirement is significantly lower than the total amount available on contemporary GPUs. Thus, the current limitation on parameter sets is primarily governed by the CUDA-imposed constraint on the number of threads within a thread block.

3.2 GPU-accelerated FHEW/TFHE Bootstrapping Implementation with FFT Foundation

When employing FFT to accelerate polynomial multiplication in larger parameter sets, particularly in scenarios such as functional bootstrapping, it is essential to consider the additional error introduced by the precision loss of floating-point operations. The utilization of IEEE double-precision floating-point numbers inevitably results in the loss of some fractional bits during the multiplication of two double numbers. This issue is exacerbated when dealing with a larger Q, such as the 54-bit modulus used in large-precision functional bootstrapping [LMP22].

To address this concern, we refine the previous empirical error analysis metric proposed by Micciancio et al. [MP21] by incorporating the supplementary FFT bias (Δ_{FFT}) introduced during FFT-based bootstrapping. This updated error analysis metric provides a more accurate assessment, particularly in large modulus bootstrapping scenarios with FFT. Furthermore, to quantify the impact of this additional error, we compute the corresponding decryption failure probabilities for existing parameter sets [MP21, LMP22]. Subsequently, we propose concrete parameter sets tailored for both gate bootstrapping and large-precision functional bootstrapping using FFT.

Error analysis. We begin by revising the error analysis metric proposed in [MP21] to account for the additional error introduced by FFT. The standard deviation of the refreshed

ciphertext (β) is calculated as follows:

$$\beta = \sqrt{\frac{q^2}{Q_{ks}^2}(\frac{Q_{ks}^2}{Q^2}\sigma_{ACC}^2 + \sigma_{MS_1}^2 + \sigma_{KS}^2) + \sigma_{MS_2}^2}.$$

Since the error caused by FFT occurs only in the update phase of bootstrapping, we add the FFT bias to σ_{ACC}^2 to derive the biased standard deviation. The estimated error for σ_{ACC}^2 is given by:

$$\sigma_{ACC}^2 = (\sigma_{ACC-GINX} + \Delta_{FFT})^2,$$

where $\sigma_{ACC-GINX}$ needs to be adjusted since our GPU implementation incorporates the approximate gadget decompose method outlined in [LMK+23]. This method involves dropping the last digit during gadget decomposition, leading to slight differences in the error analysis metric. Here we modify d_g to $d_g - 1$ since the last digit is discarded during bootstrapping, and add an additional term $\mathsf{Var}(\mathsf{t_0} \cdot \mathsf{m})$ from [LMK+23], representing the information loss introduced by discarding the last digit. The modified formula is:

$$\sigma_{ACC-GINX} = \sqrt{2u(d_g - 1)nN\frac{B_g^2}{6}\sigma^2 + 2un \cdot \mathsf{Var}(\mathsf{t_0} \cdot \mathsf{m})},$$

where $\mathsf{t_0} \in R_Q$, and $\mathsf{t_0} < B_g$.

Δ_{FFT} is the FFT bias after the LWE decryption $(b - \langle \mathbf{a}, \mathbf{s} \rangle)$, for a LWE ciphertext $(\mathbf{a}$,b) after the extract phase in bootstrapping. e_{FFT} is the FFT bias of the element in the extracted LWE ciphertext. The Δ_{FFT} is derived as the following formula:

$$\Delta_{FFT} = |(1 - \|s_n\|)e_{FFT}|,$$

where $\|s_n\| \leq \sqrt{n/2}$ as noted in [MP21]. e_{FFT} is derived by calculating the accumulated error during the bootstrapping process. The term $S \times \epsilon$ is from [DM15] represents the relative error from floating point precision loss in multiplying two polynomials during bootstrapping. To derive the overall FFT bias, we calculate the number of polynomial multiplications during the bootstrapping process, which is $2und_g$. The error generated in each polynomial can be modeled as independent Gaussian noise, so the overall noise from these operations has a square root dependence as in the following formula:

$$e_{FFT} = \sqrt{n}\lceil \sqrt{2ud_g} \times S \times \epsilon \rfloor,$$

where $\lceil \cdot \rfloor$ denotes the rounding operation from floating point to integer after IFFT, $S = B_g Q\sqrt{N}/4$ as noted in [DM15], and ϵ represents the relative error for each floating point operation, dependent on the FFT library. Δ_{FFT} represents the upper bound of the FFT bias acquired from experiment, so the choice of FFT library influences its value. Specifically, our experiments utilize the cuFFTDx library [NVI23] for FFT operations.

We verify the additional Δ_{FFT} term between NTT-based bootstrapping and FFT-based bootstrapping through comprehensive experiments. After undergoing the update phase in bootstrapping, we compare two ciphertexts, one using FFT and the other using NTT. We directly decrypt both ciphertexts and measure the difference between them. This process is repeated for a sample size of 1024 runs. The results of our experiments show that the differences between the two ciphertexts align with the additional bias term Δ_{FFT} in our revised metric. This empirical validation confirms the validity of our metric and its ability to quantify the differences between NTT-based and FFT-based bootstrapping approaches.

Concrete parameters. To evaluate the impact of FFT-based bootstrapping on decryption failure probability (FP), we recalculated the FP for each parameter set used in gate bootstrapping in [MP21] and the functions in large-precision functional bootstrapping [LMP22]. The β_{exp} values in Table 3 and Table 4 were derived from a substantial sample size of

16,384 bootstrapping runs. Consistent with prior methodologies [DM15, CGGI16, MP21], we set the upper bound for the FP to 2^{-32} for a parameter set to be considered suitable for practical computations.

In Table 3, we present the parameter sets proposed by Micciancio et al. [MP21]. The FP is calculated using the formula proposed in [DM15, MP21], which is estimated as $1-\text{erf}(\frac{q/8}{2\beta_{\text{exp}}})$. FHEW/TFHE gate bootstrapping supports gates such as AND, OR, NAND, NOR, and others. We chose the NAND gate to calculate the decryption failure probability for all parameter sets. Notably, most of the decryption failure probabilities are lower than the values listed in [MP21]. This difference arises because the plaintext space used in gate bootstrapping is relatively small, only four, which easily satisfies the condition $\mathsf{Var}(\mathsf{m}) \leq \sigma^2$ mentioned in the approximate gadget decompose method, resulting in even lower decryption failure probabilities.

Table 3: Parameter set and the corresponding decryption failure probability (FP) for our FFT-based gate bootstrapping implementation.

Parameter Set	n	q	N	log_2Q	log_2Q_{ks}	B_{ks}	B_g	FP	β_{exp}
STD128	512	1024	1024	27	2^{14}	2^7	2^7	2^{-54}	10.72
STD192	1024	1024	2048	37	2^{19}	28	2^{13}	2^{-107}	7.55
STD256	1024	2048	2048	29	2^{14}	2^7	2^8	2^{-33}	27.96
STD128Q	1024	1024	2048	50	2^{25}	2^5	2^{25}	2^{-107}	7.56
STD192Q	1024	1024	2048	35	2^{17}	2^6	2^{12}	2^{-97}	7.93
STD256Q	2048	2048	2048	27	2^{16}	2^4	2^7	2^{-58}	20.81

For large-precision functional bootstrapping [LMP22], the corresponding FP are presented in Table 4. These values are calculated using the formula introduced in [LMP22]. The term σ_{sum} in the original formula calculates the sum of two independent ciphertexts; therefore, σ_{sum} is essentially $\sqrt{2}\beta_{\text{exp}}$. The formula can then be expressed as $1-\text{erf}(\frac{q/p}{4\beta_{\text{exp}}})$.

Table 4: Parameter set and the corresponding decryption failure probability (FP) for our FFT-based Large-precision functional bootstrapping [LMP22] implementation, with fixed parameters $log_2Q = 54, N = 2048, n = 1305, log_2Q_{ks} = 35, B_{ks} = 32$. Q' stands for ciphertext modulus in [LMP22]

Function	q	Q'	$\log P$ [bits]	B_g	FP	β_{exp}
HomSign/DigitDecomp	4096	2^{12}	4	2^{27}	2^{-68}	9.52
HomSign/DigitDecomp		2^{16}	8	2^{27}	2^{-67}	9.59
HomSign/DigitDecomp		2^{20}	12	2^{18}	2^{-79}	8.85
HomSign/DigitDecomp		2^{24}	16	2^{18}	2^{-88}	8.34
HomSign/DigitDecomp		2^{25}	17	2^{18}	2^{-85}	8.51
HomSign/DigitDecomp		2^{26}	18	2^{14}	2^{-82}	8.65
HomSign/DigitDecomp		2^{28}	20	2^{14}	2^{-86}	8.45
HomSign/DigitDecomp		2^{29}	21	2^{14}	2^{-83}	8.63
EvalFunc	2048	2^{12}	3	2^{27}	2^{-69}	9.5

4 Performance Evaluation

We evaluated the performance of our GPU-accelerated FHE library by benchmarking various parameter sets, bootstrapping methods, and comparing it with the state-of-the-art library. In Section 4.1, we implement the original gate bootstrapping described in [DM15, CGGI16] and the functional bootstrapping as proposed by Liu et al. [LMP22]. To show the scalability of our library, we benchmark these functions using multiple GPUs, comparing them with a powerful 64-thread CPU. In Section 4.2, we compare the performance of single-block bootstrapping and multi-blocks bootstrapping across three

similar parameter sets with different B_g, where the value balances the running time and the noise growth rate when performing the encrypted operations. Each of these bootstrapping realizations exhibits preferred performance domains in specific parameter ranges. Finally, in Section 4.3, we compare our GPU implementation with TFHE-rs [Zam22], the state-of-the-art FHEW/TFHE GPU-accelerated library. All the runtime performances reported in this section are end-to-end runtime, including CPU runtime, GPU runtime, and data transfer time between GPU and CPU. Our results demonstrate performance advantages in most scenarios and offer a more flexible configuration to meet different user scenarios.

Testbed. We implement our library upon the OpenFHE library version 1.0.4 with its default compiler settings and 64-bit integer. The benchmarking is conducted on a CPU server equipped with the AMD Ryzen Threadripper 3970X CPU (32 cores, 64 threads) and 128GB RAM. We employ multiple NVIDIA RTX4090 GPUs (24GB VRAM) for the GPU-accelerating experiments. The operating system is Ubuntu 22.04.2 LTS. In CPU benchmarking, OpenMP is utilized for parallel computation, where each thread accounts for one bootstrapping operation.

4.1 The Running Time of FHEW/TFHE Bootstrapping

The performance evaluation of bootstrapping functions encompasses all functions available in the TFHE/GINX scheme in the OpenFHE library, including EvalBinGate, EvalFunc, EvalFloor, EvalSign, and EvalDecomp. We measure the execution time in milliseconds required for evaluating a ciphertext. We compare all functions on a CPU with 1 thread, a CPU with 64 threads, 1 GPU supported, and 8 GPU-supported. This comparison aims to demonstrate the substantial speedup achieved by our GPU implementation in a scenario where both CPU and GPU are fully utilized.

Table 5 shows the evaluation results with a batch size 16384. All parameter settings for functional bootstrapping in FHE are aligned with those in [LMP22]. For EvalFunc, we choose the function to be non-negacyclic for fair comparison. In this case, the EvalFunc evaluation will execute two bootstrappings in its core. For EvalBinGate, we evaluate the runtime of the NAND gate using the security level of STD128. When using a single GPU, we observe that the speedup for various types of bootstrapping is approximately 18× compared to the CPU utilizing 64 threads. The slight variances between different bootstrappings arise from differences in CPU workload distribution.

When increasing the number of GPUs, we observe a ∼7× speedup when using 8 GPUs compared to 1-GPU bootstrapping functions experiments. The result indicates that the overhead from the CPU portion is minor. Figure 4 provides detailed speedup for each bootstrapping function with different numbers of GPUs. We observe that the speedup when using multiple GPUs in EvalBinGate is smaller than other bootstrapping functions. This is because of the parameter set of EvalBinGate, which is relatively small, resulting in

Table 5: Execution time of FHEW/TFHE bootstrapping using 1 CPU thread (CPU1T), 64 CPU threads (CPU64T), and our GPU implementation including 1 GPU and 8 GPU

	Execution Time (ms/ctx)				
	EvalBinGate	**EvalFunc**	**EvalFloor**	**EvalSign**	**EvalDecomp**
n	512	1305	1305	1305	1305
q	1024	2048	2048	4096	4096
$log_2 Q$	27	54	27	54	54
N	1024	2048	1024	2048	2048
B_g	2^7	2^{27}	2^5	2^{18}	2^{18}
CPU1T	153 (1×)	1037 (1×)	1158 (1×)	3424 (1×)	4092 (1×)
CPU64T	4.3 (36×)	28.5 (36×)	34.6 (34×)	93.6 (37×)	112.6 (36×)
1 GPU	0.23 (665×)	1.59 (652×)	1.68 (689×)	4.99 (686×)	6.03 (679×)
8 GPU	0.04 (3825×)	0.23 (4509×)	0.23 (5035×)	0.69 (4962×)	0.88 (4650×)

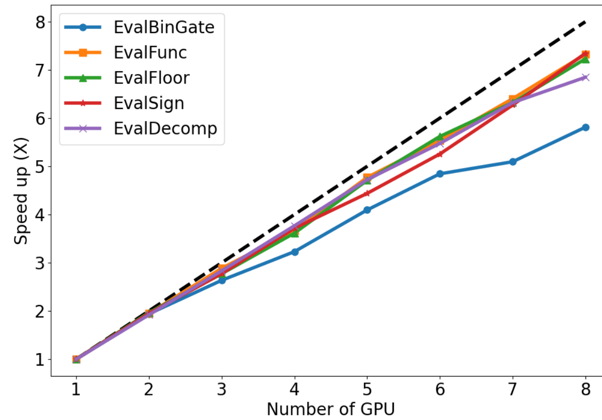


Figure 4: The speed up of different bootstrapping functions with different number of GPUs compared to the corresponding single-GPU experiment.

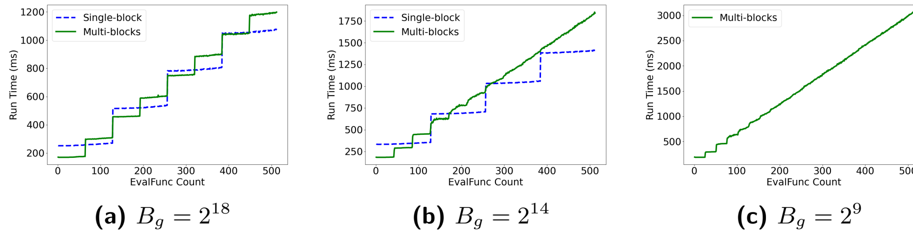
a larger CPU workload rate compared to the other functions. Therefore, the benefit of involving more GPUs is relatively small (from ∼7× to ∼6×). Overall, our method only shows a small decline. Furthermore, our design allows users to freely determine the number of GPUs, which shows strong scalability for real-world applications.

4.2 Single-Block Bootstrapping vs. Multi-Blocks Bootstrapping

We compare single-block and multi-blocks bootstrappings on three different B_g values to show the effectiveness of our implementation. Figure 5 presents the performance evaluations on EvalFunc with varying B_g values. Notably, a larger B_g corresponds to a smaller GPU configuration—specifically, fewer threads and reduced shared memory. Single-block bootstrapping exhibits a ladder-shaped runtime as the number of EvalFunc increases. This occurs because each bootstrapping operation is executed in one Streaming Processor (SM) of the GPU. When the number of bootstrapping operations is less than or equal to the total SMs in the GPU, the runtime remains constant. Exceeding this number doubles the runtime. Conversely, in multi-block bootstrapping, one bootstrapping operation may be split into multiple thread blocks, distributed across multiple SMs by the GPU runtime scheduler.

The required number of threads per thread block for different B_g values is calculated using the formula introduced in Section 3.1. In all three figures presented in Figure 5, the value of $\mathsf{elem}_{\mathsf{FFT}}$ is constant at 8. In the case of a large B_g value, single-block bootstrapping necessitates 512 threads in 1 thread block, while multi-blocks bootstrapping requires 256 threads distributed across two thread blocks. For a medium B_g value, single-block bootstrapping demands 768 threads in 1 thread block, while multi-blocks bootstrapping requires 256 threads distributed across three thread blocks. In the case of a small B_g value, single-block bootstrapping becomes infeasible, and multi-blocks bootstrapping requires 256 threads distributed across 5 thread blocks.

It is evident that if the number of threads per block is significantly less than the CUDA limitation of threads per block (as seen in Figure 6a), multi-blocks bootstrapping exhibits superior performance, as a single Streaming Multiprocessor (SM) efficiently handles all the thread blocks assigned to it. However, as the number of threads per block approaches the CUDA limitation (as seen in Figure 6b), single-block bootstrapping performs better than multi-blocks bootstrapping. This phenomenon is because the CUDA scheduler may distribute some blocks from multi-blocks bootstrapping to other SMs. Finally, only multi-blocks bootstrapping effectively handle such small B_g value when the number of threads



(a) $B_g = 2^{18}$ **(b)** $B_g = 2^{14}$ **(c)** $B_g = 2^9$

Figure 5: Run time comparison of EvalFunc in three different B_g using single-block and multi-blocks bootstrapping with $n = 1305$, $q = 2048$, $N = 2048$, and $log_2 Q = 54$. A high B_g value has a short FHEW/TFHE bootstrapping running time.

per block surpasses the CUDA limitation (as observed in Figure 6c).

4.3 Comparison with TFHE-rs GPU Backend

We evaluate the performance of our implementation and TFHE-rs on the same application, specifically the evaluation of an AND gate for two integers homomorphically. We follow the benchmarking methodology outlined on the TFHE-rs website. For instance, `uint16` indicates the evaluation of the AND gate between two sixteen-bit unsigned integers. The FHEW/TFHE scheme has many variants, which may differ in bootstrapping algorithms, encryption/decryption methods, or FHE parameter selection. As outlined in Section 2.1, we adopt the enhanced GINX method proposed by Bonte et al. [BIP+22], also known as TFHE/GINX in [BBB+22]. TFHE-rs has its own TFHE variants [CJP21]. The TFHE variants used in TFHE-rs differ in encryption/decryption methods, bootstrapping algorithms, and FHE parameters from our implementation. For instance, TFHE-rs encrypts two bits in one ciphertext with larger values of Q and N during AND gate evaluation, while we encrypt one bit in one ciphertext with smaller values of Q and N. These differences result in different numbers of bootstrappings and operations in each implementation.

For a fair comparison, we ensure that both implementations have the same security level and comparable decryption failure probability. In the TFHE-rs benchmark, we use the default parameter set provided in the example of running the AND gate on the GPU on its website, which offers a security level of 128 bits estimated by the lattice estimator [APS15], with a decryption failure probability of 2^{-40}. For our implementation, we also utilize a standard 128-bit security level for the parameter set detailed in Table 3. We slightly adjust n to 502 to achieve faster bootstrapping times while maintaining the same security level. The decryption failure probability for this parameter set is 2^{-56}.

In Table 6, we compare the runtime of our implementation with TFHE-rs library version 0.5.3. Across most scenarios (from Uint8 to Uint128), our implementation exhibits superior latency compared to TFHE-rs. Notably, we utilize our multi-blocks bootstrapping approach for scenarios ranging from `uint8` to `uint64`, highlighting the efficiency of this strategy when the number of bootstrapping is not large. Even after transitioning to single-block bootstrapping beyond `uint64`, our implementation continues to demonstrate improved performance. In the evaluation of `uint256`, our implementation offers comparable performance with TFHE-rs.

Our advantages over TFHE-rs. In addition to the performance advantages, our implementation offers greater flexibility for various scenarios, from general use to research purposes, in two key aspects. Firstly, TFHE-rs provides predefined choices for the number of bootstrappings to concurrently run on GPU (e.g., 8, 16, 32, etc.). Our implementation allows users to configure an arbitrary number of bootstrappings according to their specific workload requirements. This enables applications such as secure neural network inference to dynamically adjust the number of parallel bootstrappings based on the number of activation function nodes, which may vary in different network structures. Secondly,

Table 6: Latency (ms) comparison of TFHE-rs with our implementation evaluating AND gate using different integer types, e.g. `uint8` means the 8-bit unsigned integer.

	`uint8`	`uint16`	`uint32`	`uint64`	`uint128`	`uint256`
TFHE-rs	31.53	31.54	31.55	32.03	33.74	58.32
Ours	18.63	18.61	18.87	24.23	29.97	58.3
Speedup	1.69×	1.69×	1.67×	1.32×	1.13×	1×

our method allows ciphertexts with different lookup tables concurrently running on the GPU. This functionality is particularly valuable for applications such as private decision tree evaluation, where ciphertexts with different conditional statements can be executed simultaneously. These two aspects of flexibility make our GPU implementation a more suitable tool for researchers to test and develop their FHE applications, offering greater adaptability to diverse research scenarios and requirements.

Compared to other works. Our GPU solution, when compared with the current frontier GPU solution, TFHE-rs, which significantly outperforms previous GPU-based implementations [ver18, nuc18] in speed, highlights our advancements in GPU-based solutions.

Additionally, we compare our implementations with two other hardware-based solutions: an ASIC-based approach, MATCHA [JLJ22], and an FPGA-based approach, FPT [vDTV23]. Notably, the programmable bootstrapping (PBS) methods used in these two works implicitly constrain the input function to be negacyclic. Their PBS algorithm is essentially equivalent to single gate bootstrapping, making it directly comparable to our EvalBinGate method, as shown in Table 5, which also involves single bootstrapping. In contrast, large-precision functional bootstrapping operations, such as EvalFunc in Table 5, impose no such constraints on input functions, often requiring multiple bootstrapping operations, and are therefore slower.

In summary, while MATCHA and FPT are faster than our GPU solutions when comparing their PBS to our EvalBinGate, their implementations rely on somewhat outdated FHE parameters from older versions of TFHE. These parameters either provide only 110-bit security or 128-bit security with a 2^{-25} decryption failure probability. Specifically, our EvalBinGate achieves 4.35 executions per millisecond, with 128-bit security and a failure probability of 2^{-54}; FPT reports a throughput of 28.4 PBS operations per millisecond on a high-end U280 FPGA with 128-bit security, but a failure probability of 2^{-25}; MATCHA achieves 10 PBS operations per millisecond in simulation, but with 110-bit security.

Currently, 128-bit security is considered the standard baseline, and failure probabilities above 2^{-32} are typically deemed unacceptable in practice [DM15, CGGI16, MP21].

We believe that conducting a fair comparison between our GPU solution and scaled-up versions of MATCHA and FPT would be valuable, though it would require substantial effort beyond the scope of this work. MATCHA is ASIC-based, and optimizing for different parameters could involve significant additional work. FPT, which uses fixed-point calculations for FFT-based implementations, would require a detailed noise analysis for each parameter set, meaning parameter changes could necessitate redoing much of their optimization process. We leave as an important and interesting open question the evaluation of ASIC/FPGA-based solutions using more up-to-date parameters.

5 Applications

In this section, we demonstrate the versatility of our GPU implementation for three distinct applications. Firstly, in Section 5.1, we benchmark commonly used non-polynomial functions in neural networks, such as ReLU and maxpooling. These functions are essential for applying FHE to neural network applications. Next, we demonstrate these non-polynomial functions by running a secure neural network inference using the MNIST dataset in Section 5.2, which is a critical application in the machine learning field. Lastly,

in Section 5.3, we explore a classical machine learning application, a decision tree, with input data encrypted using FHE. By leveraging our GPU implementation, we achieve real-time evaluation with minimal latency.

5.1 Secure Non-polynomial Functions for Neural Networks

In the domain of neural networks, functions like Rectified Linear Unit (ReLU), max-pooling, and sigmoid activation serve as fundamental building blocks within the network architecture, contributing significantly to its overall performance. Consider a typical Convolutional Neural Network (CNN), where numerous activation nodes need simultaneous evaluation. Here, the throughput of evaluating these non-polynomial functions becomes critical. Poor throughput can significantly impact inference time.

We conduct benchmark on these commonly used non-polynomial functions, measuring the runtime of these functions. For functions like ReLU, sigmoid, square root, and reciprocal, it is essentially equivalent to running a single EvalFunc function with customized LUT. Since these functions are all non-negacyclic, the runtime of these functions will follow the benchmark in Table 5. For the max-pooling function, our FHE implementation follows the max-tree implementation outlined in Pegasus [jLHH+21], which needs 4 LUT evaluations for a 2X2 max-pooling. For our experiments, we follow the default parameters of the EvalFunc as detailed in Table 4,

We used a single NVIDIA RTX4090 GPU to benchmark the runtime of our implementation. The benchmark results are compared with the CPU runtime using 20 threads from Pegasus [jLHH+21] in Table 7. Remarkably, our implementation showcases a significant 30× throughput enhancement across all non-polynomial functions compared to CPU runtime. This substantial acceleration signifies the potential of our implementation to expedite applications requiring evaluating a large number of these functions, thus offering substantial performance gains.

Table 7: Throughput comparison of secure non-polynomial functions. [jLHH+21] utilized a 20-thread CPU.

Approach	Throughput	
	sigmoid/ReLU/sqrt/reciprocal	**Max-Pooling (2x2)**
[jLHH+21]	20.77/s (1×)	4.83/s (1×)
Ours	628.93/s (30.3×)	147.7/s (30.6×)

5.2 Secure Neural Network Inference

FHE-based encrypted neural network inference presents a promising solution for secure and privacy-preserving machine learning, facilitating collaborative research and data sharing while addressing concerns about data privacy and security in an interconnected world. The MNIST dataset [LCB10] has been a popular choice for evaluating FHE-based secure inference. Several research efforts have focused on applying FHE to achieve secure inference on this dataset [LLZ+24, BMMP18, FGT21, KS23].

In evaluating the FHE-based encrypted neural network inference on the MNIST dataset, we categorize the model architectures of existing works into two categories: small and large. For all the model architectures, one will have the same input layer with 784 (28x28) neurons and the same output layer with 10 neurons. The differences lie in the hidden layer configurations as shown in Table 8. To ensure a fair comparison, we train our small model with one hidden layer containing 30 neurons. For the large model, we train it with three hidden layers, each containing 1024 neurons. This approach allows us to evaluate our FHE-based encrypted neural network inference solution against existing

works using comparable model architectures. For the model training and inference, we apply another concurrent work on FHE-based encrypted neural network training and fine-tuning [KLX+24]. Improving the accuracy of the FHE-based encrypted neural network inference is challenging and is the contribution of the work [KLX+24].

The dataset consists of 60000 instances, of which we randomly selected 50000 for training and the remaining 10000 for testing. In all our experiments, we employed the EvalFunc to realize the ReLU function. We made slight adjustments to the default parameters as outlined in Table 4, setting $p = 2^{16}$ to accommodate the input size of the MNIST dataset and $B_g = 2^{18}$ to accommodate additional noise. For the fully connected layer, we employ the cuBLAS library [NVI24] to perform matrix operations on the GPU. The experiment was conducted using 2 NVIDIA RTX4090 GPUs to ensure a comparable computing power with the GPU work RED [FGT21], which utilized 8 NVIDIA T4 GPUs in their experiment.

In Table 8, we compared both small and large model architectures with the existing works. Regarding runtime performance, our implementation outperformed FHE-DiNN [BMMP18] with a 12× speedup and [LLZ+24] with a 3.5× speedup. Additionally, compared to both works, we achieved a higher security level of 128 bits, whereas both implementations only attained 80 bits. An 80-bit security level parameters run at least 2× faster than the parameters with a 128-bit security level in the same testing environment due to a smaller polynomial dimension. Compared to the large model architecture, our implementation demonstrates a 456× speedup compared to FDFB [KS23] while achieving a higher security level and a bigger network structure, and a 1.37× speedup compared to RED [FGT21] with the same network structure and comparable computing power.

Table 8: Comparison of existing Secure Neural Network Inference on MNIST. The hidden layer CXD means C layers with D neurons per layer

Approach	Security Level	Activation Function	Hidden Layer	Runtime (s)	Accuracy (%)	
					HE	Plaintext
[BMMP18]	80 bits	Sign	1X30	0.49	93.71	94.76
[LLZ+24]	80 bits	ReLU	1X30	0.14	94.04	94.80
Ours	128 bits	ReLU	1X30	**0.04**	96.47	97
[FGT21]	128 bits	ReLU	3X1024	8.2†	99	-
[KS23]	100 bits	ReLU	1X510	2736	95	-
Ours	128 bits	ReLU	3X1024	**6**	97.2	97.8

† [FGT21] utilized 8 NVIDIA T4 GPUs.

5.3 Private Decision Tree Evaluation

Decision trees are widely utilized in machine learning and data analysis for classification and regression tasks. These tree-like structures feature internal nodes representing decisions based on specific features and leaf nodes denoting the corresponding outcomes or class labels. Leveraging FHE to safeguard user data privacy has been extensively explored in the literatures [LZS18, TBK20, jLHH+21].

We follow the FHE decision tree algorithm described in [jLHH+21]. This algorithm evaluates $\mathcal{O}(2N)$ LUT functions, where N is the number of nodes in the decision tree. Notably, these LUT functions require the ability to concurrently compute ciphertexts with different lookup tables, demonstrating the flexibility of our GPU implementation. For our experiments, we slightly modified the default parameters of the EvalFunc as detailed in Table 4, with $p = 2^{11}$ to accommodate the input of the decision tree and $B_g = 2^9$ to accommodate more noise.

We utilized one NVIDIA RTX4090 GPU during the experiment and adopted the multi-blocks bootstrapping approach. We evaluated our private decision tree on the

Iris dataset [Fis88] from the UCI repository. The dataset consists of 150 instances, of which we randomly selected 100 instances for training and the remaining 50 instances for testing. In Table 9, we compare the runtime latency (cloud side), communication overhead, and accuracy of both the HE and plaintext between our method and three existing works [LZS18, TBK20, jLHH+21]. All these works utilize a 16-thread CPU to execute multiple LUT evaluations concurrently. Among them, our approach achieves the fastest latency, enabling real-time evaluation. For communication, we need to send 4 LWE ciphertexts (4 features) to the cloud and 1 LWE ciphertext (result) back to the user. It is noteworthy that while [TBK20] managed to maintain accuracy levels comparable to plaintext classification, it incurred a substantial 24× communication overhead and operated 2.47 times slower than our method.

Table 9: Comparison of Private Decision Tree Evaluation on Iris dataset, which has ≈ 10 internal nodes, 4 features, and 3 classification labels

Approach	Communication	Latency (s)	Accuracy (%)	
			HE	Plaintext
[jLHH+21]	16.89 KB	1.87	94.74	97.37
[TBK20]	1.19 MB	0.94	97.37	97.37
[LZS18]	1.65 MB	0.59	95.33	97.37
Ours	51.01 KB	**0.38**	98	100

6 Conclusion

In this paper, we introduced an innovative GPU parallelization strategy used to tackle the bottleneck operation, bootstrapping, in FHEW/TFHE schemes. Previous works encountered challenges due to the limited flexibility of parameters, making it difficult to extend them to functional bootstrapping requiring larger parameter configurations. Through our novel strategy, we successfully broadened the scope of the supported parameters, supporting bootstrapping from gate bootstrapping to large-precision functional bootstrapping and providing flexibility across security levels (STD128 to STD256). We demonstrate the versatility of our approach through its applicability to numerous real-world scenarios, including private decision tree evaluation and secure neural network inference. The flexibility and efficiency of our method enables new potentials for practical FHE-based privacy-preserving solutions.

Future work. Future endeavors include exploring the utilization of NVIDIA's GPU tensor core [MCL+18] to further accelerate bootstrapping. Tensor cores exhibit higher throughput in matrix multiplication, showcasing immense potential. Additionally, it is possible to extend our work to support SIMD-type solutions for the third-generation FHE (e.g., [MS18, LW23a, LW23b, GPV23, MKMS24]), exploring how to accelerate these theoretical methods.

Acknowledgments

This work was primarily conducted at Inventec Corporation. We gratefully acknowledge the support of the National Science and Technology Council, Taiwan, under grant NSTC 112-2221-E-002 -159 -MY3 and 113-2634-F-002-001 -MBK. Feng-Hao Liu would like to thank NSF Career Award CNS-2402031. Their support was instrumental in developing critical preliminary results for this work. We also extend our sincere thanks to the anonymous reviewers for their valuable feedback, which significantly improved the quality of this paper.

References

[AP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 297–314. Springer, Berlin, Heidelberg, August 2014.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[BBB+22] Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, R. V. Saraswathy, Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles, CA, USA, 7 November 2022*, pages 53–63. ACM, 2022.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[BIP+22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 188–215. Springer, Cham, December 2022.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). `https://github.com/intel/hexl`, september 2021.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 483–512. Springer, Cham, August 2018.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Berlin, Heidelberg, August 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 505–524. Springer, Berlin, Heidelberg, August 2011.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Lattice-based FHE as secure as PKE. In Moni Naor, editor, *ITCS 2014*, pages 1–12. ACM, January 2014.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Berlin, Heidelberg, December 2016.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 360–384. Springer, Cham, April / May 2018.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings*, volume 12716 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Cham, December 2017.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 617–640. Springer, Berlin, Heidelberg, April 2015.

[FGT21] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. REDsec: Running encrypted DNNs in seconds. Cryptology ePrint Archive, Report 2021/1100, 2021. https://eprint.iacr.org/2021/1100.

[Fis88] R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

[FJ98] Matteo Frigo and Steven G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98, Seattle, Washington, USA, May 12-15, 1998*, pages 1381–1384. IEEE, 1998.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[FWX+23] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. Tensorfhe: Achieving practical computation on encrypted data using GPGPU. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 922–934. IEEE, 2023.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, USA, 2009.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[GPV23] Antonio Guimarães, Hilder V. L. Pereira, and Barry Van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*, pages 3–35. Springer, Singapore, December 2023.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Berlin, Heidelberg, August 2013.

[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 TCHES*, 2021(4):114–148, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9062.

[jLHH+21] Wen jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy*, pages 1057–1073. IEEE Computer Society Press, May 2021.

[JLJ22] Lei Jiang, Qian Lou, and Nrushad Joshi. MATCHA: a fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 235–240. ACM, 2022.

[Kle21] Jakub Klemsa. Fast and error-free negacyclic integer convolution using extended fourier transform. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings*, volume 12716 of *Lecture Notes in Computer Science*, pages 282–300. Springer, 2021.

[KLX+24] Yu-Te Ku, Feng-Hao Liu, Yu Xiao, Ming-Ching Chang, Chih-Fan Hsu, I-Ping Tu, Shih-Hao Hung, and Wei-Chao Chen. Efficient third generation fhe based non-interactive encrypted dnn inference with fhe-aware training. In *Unpublished Manuscript (2024)*, 2024.

[KS23] Kamil Kluczniak and Leonard Schild. FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR TCHES*, 2023(1):501–537, 2023.

[LCB10] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[LLZ+24] Kwok-Yan Lam, Xianhui Lu, Linru Zhang, Xiangning Wang, Huaxiong Wang, and Si Qi Goh. Efficient fhe-based privacy-enhanced neural network for trustworthy ai-as-a-service. *IEEE Trans. Dependable Secur. Comput.*, 21(5):4451–4468, 2024.

[LMK+23] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 227–256. Springer, Cham, April 2023.

[LMP22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 130–160. Springer, Cham, December 2022.

[LW23a] Feng-Hao Liu and Han Wang. Batch bootstrapping I: A new framework for SIMD bootstrapping in polynomial modulus. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 321–352. Springer, Cham, April 2023.

[LW23b] Feng-Hao Liu and Han Wang. Batch bootstrapping II: Bootstrapping in polynomial modulus only requires $\tilde{O}(1)$ FHE multiplications in amortization. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 353–384. Springer, Cham, April 2023.

[LZS18] Wenjie Lu, Jun-Jie Zhou, and Jun Sakuma. Non-interactive and output expressive private comparison from homomorphic encryption. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 67–74. ACM Press, April 2018.

[MCL+18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 522–531. IEEE Computer Society, 2018.

[MKMS24] Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized FHEW bootstrapping using ring automorphisms. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part II*, volume 14604 of *LNCS*, pages 322–353. Springer, Cham, April 2024.

[MP21] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in fhew-like cryptosystems. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, Korea, 15 November 2021*, pages 17–28. WAHC@ACM, 2021.

[MS18] Daniele Micciancio and Jessica Sorrell. Ring packing and amortized FHEW bootstrapping. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 100:1–100:14. Schloss Dagstuhl, July 2018.

[nuc18] nucypher. A gpu implementation of fully homomorphic encryption on torus. https://github.com/nucypher/nufhe, 2018. accessed: 2024-01-09.

[NVI23] NVIDIA Corporation. NVIDIA cuFFTDX Library. https://docs.nvidia.com/cuda/cufftdx/index.html, 2023. accessed: 2024-01-09.

[NVI24] NVIDIA Corporation. NVIDIA cuBLAS Library. https://developer.nvidia.com/cublas, 2024. accessed: 2024-06-09.

[TBK20] Anselme Tueno, Yordan Boev, and Florian Kerschbaum. Non-interactive private decision tree evaluation. In Anoop Singhal and Jaideep Vaidya, editors, *Data and Applications Security and Privacy XXXIV - 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25-26, 2020, Proceedings*, volume 12122 of *Lecture Notes in Computer Science*, pages 174–194. Springer, 2020.

[vDTV23] Michiel van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 741–755. ACM Press, November 2023.

[ver18] vernamlab. Cuda-accelerated fully homomorphic encryption library. https://github.com/vernamlab/cuFHE, 2018. accessed: 2024-01-09.

[WFF11] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21(1):18749–19424, 2011.

[Zam22] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.

[ZCY+22] Junxue Zhang, Xiaodian Cheng, Liu Yang, Jinbin Hu, Ximeng Liu, and Kai Chen. Sok: Fully homomorphic encryption accelerators. *CoRR*, abs/2212.01713, 2022.