Offloading Key Switching on GPUs: A Path towards Seamless Acceleration of FHE

Orion Papadakis, Michail Papadimitriou, Athanasios Stratikopoulos, Maria Xekalaki,
Juan Fumero and Christos Kotselidis
Department of Computer Science
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

***Abstract*—Fully Homomorphic Encryption (FHE) enables secure computations on encrypted data, offering strong privacy guarantees for cloud computing, privacy-preserving machine learning, and confidential data processing. However, the computational overhead associated with FHE operations, due to the large size of ciphertext and the high arithmetic complexity, limits its practical applicability.**

In this work, we address this challenge by presenting an approach that is implemented within the OpenFHE library in order to offload the most dominant components of key switching for the BGV scheme on GPU hardware. In particular, the scope of this work is the performance improvement of the Approximate Modulus Downscaling (ApproxModDown) function. Our experimental evaluation shows that the proposed system can yield up to a 4.58× performance speedup against the vanilla OpenFHE ApproxModDown implementation, while also resulting in 1.16× performance improvement per homomorphic multiplication and 1.08× improvement for end-to-end execution time.

***Index Terms*—data privacy, fully homomorphic encryption, hardware acceleration, GPUs**

I. Introduction

Fully Homomorphic Encryption (FHE) enables computations on encrypted data without requiring decryption, offering strong privacy guarantees for secure cloud computing, privacy-preserving machine learning, and confidential data processing [1]–[3]. However, FHE operations incur significant computational overhead due to the large size of ciphertext and the high arithmetic complexity involved in homomorphic computations. As a result, efficient implementations that utilize hardware acceleration are critical for making FHE practical.

Among the various FHE schemes, Brakerski-Gentry-Vaikuntanathan (BGV) is widely used for its efficiency in supporting leveled homomorphic computations [4]. One of the key challenges in FHE schemes like BGV is noise growth, which accumulates as long as the number of homomorphic operations grows, eventually posing the need for noise management techniques such as key switching. For example, the *Approximate Modulus Downscaling* (ApproxModDown) function plays a crucial role in this process, as it enables ciphertext modulus reduction while maintaining correctness.

Existing FHE libraries, such as OpenFHE [5], Microsoft SEAL [6], and Zama's TFHE-rs [7], provide software implementations of homomorphic operations optimized for general-purpose CPUs. While these libraries incorporate algorithmic optimizations, they do not typically support GPU acceleration for computationally intensive tasks. Prior works have explored GPU implementations of FHE operations, but many of them often target schemes other than BGV, or develop standalone GPU-based FHE libraries.

In this work, we present a novel approach that provides GPU acceleration for the key switching operation in BGV homomorphic multiplication. The system is open-source and integrates hardware acceleration seamlessly into the OpenFHE library. Our experiments show that users can achieve up to 1.08× end-to-end performance improvement, without requiring to explicitly program the GPU. We have performed an in-depth analysis that shows that our implementation can yield up to 4.58× performance speedup when offloading the key switching operation on a GPU. Finally, our experimental evaluation shows that GPU acceleration can be useful when high-depth computations are feasible (i.e., when the computational depth of homomorphic multiplications is higher than five).

II. Background

A. Fully Homomorphic Encryption

FHE is a cryptographic technique that allows computations to be performed directly on encrypted data without requiring decryption. This property makes FHE particularly useful for privacy-preserving applications such as secure cloud computing and encrypted machine learning [3], [8]. Among various FHE schemes, Brakerski-Gentry-Vaikuntanathan (BGV) [9] is a Ring-Learning with Errors (RLWE)-based [10] leveled FHE scheme.

Practical applications of FHE often require performing multiple sequential operations on encrypted data. However, each homomorphic operation, particularly multiplications, introduces additional noise into the ciphertext. The ability to perform consecutive operations while controlling noise and ensuring successful decryption is referred to as the *computational depth* [9]. To support deep computations, BGV employs noise management techniques that mitigate noise growth and extend the computational depth.

B. Key Switching

Key switching is a noise management technique [11] that enables ciphertexts to be transformed from one key to another

while maintaining correctness and reducing noise. Without key switching, ciphertexts would accumulate excessive noise, thereby limiting the number of consecutive homomorphic operations that can be performed. Since successful decryption requires noise to remain within a certain threshold, key switching is essential for enabling deeper computations without compromising correctness.

C. Hardware Acceleration

Hardware accelerators such as, GPUs and FPGAs, enhance software performance through heterogeneous programming models like OpenCL [12], CUDA [13], and oneAPI [14]. These models expose APIs to simplify development and follow a common three-step workflow [15]: 1) transferring data from CPU memory to accelerator memory, 2) executing parallel computation, either via source code (e.g., CUDA, OpenCL, DPC++) or pre-compiled binaries (e.g., SPIR-V), and 3) moving results back to CPU memory.

While heterogeneous programming models provide APIs for hardware acceleration, achieving optimal performance is complex. Developers must manage data transfer overhead and consider hardware architecture, accelerator type, and interconnect bandwidth (e.g., PCIe). GPUs excel in fine-grained parallelism, executing thousands of threads on multiple data items [16], whereas FPGAs leverage on-chip resources for customized coarse-grain execution [17].

III. Related Work

A. FHE Libraries

Various FHE libraries implement different cryptographic schemes. OpenFHE [5] (C++) supports BGV, BFV, CKKS, DM (FHEW), and CGGI (TFHE), while Microsoft SEAL [6] (C++) focuses on BFV and CKKS. Zama's TFHE-rs [7] and Concrete [18] (Rust) specialize in TFHE, and Lattigo [19] (Go) targets cloud-based BFV and CKKS applications. Despite their strengths, none offers a standardized mechanism for seamless GPU integration, thereby requiring custom implementations for hardware acceleration.

B. Hardware Acceleration

Hardware acceleration plays a crucial role in improving the efficiency of homomorphic encryption. Various approaches have been explored, including CPU vectorization, GPU acceleration, and FPGA-based solutions.

CPU-Based Acceleration. OpenFHE includes a Hardware Abstraction Layer (HAL) that enables optimized execution of homomorphic operations. The HAL currently supports Intel's HEXL library, which utilizes Advanced Vector Extensions (AVX) to accelerate modular arithmetic. Microsoft SEAL and PALISADE also integrate HEXL to enhance their performance.

GPU Acceleration has been extensively studied for homomorphic encryption. Badawi et al. [20] proposed a parallel GPU implementation of the BFV scheme, achieving significant speedups over Microsoft SEAL. Other works have analyzed multi-threaded CPU and GPU implementations of BFV for PALISADE [21] and proposed parallel GPU implementations of leveled FHE [22]. Research has also optimized GPU implementations of the Number Theoretic Transform (NTT) to mitigate shared memory conflicts and thread divergence, notably in cuHE [23] and Microsoft SEAL [24]. HEonGPU [25] and [26] introduced an accelerated GPU implementation for Microsoft SEAL, offloading all major operations of the BFV scheme (addition, multiplication, relinearization, and rotation) to the GPU. Compared to these works, our approach targets the BGV scheme, supporting larger ring dimensions (up to $n = 65536$ versus their $n = 32768$ limit) and focusing specifically on homomorphic multiplication.

More recently, Cheddar [27] and PhantomFHE [28] have proposed fully GPU-based libraries for CKKS and BFV, CKKS, and BGV respectively. PhantomFHE achieves a $1.3\times$ speedup for BGV homomorphic multiplication, while our implementation achieves $1.16\times$ (see section VI-B). However, PhantomFHE offloads the entire homomorphic multiplication and relies on advanced algorithmic optimizations while our design selectively accelerates only 47% (see section IV) of the homomorphic multiplication, thus simplifying development by reusing pre-existing and highly optimized components of a FHE library.

FPGA Acceleration has been explored for homomorphic encryption. Intel's open-source HEXL FPGA library was archived in December 2023. Agrawal et al. [29] proposed an FPGA-based somewhat homomorphic encryption architecture, while Riazi et al. [30] introduced HEAX, a high-performance modular arithmetic engine. Sinha et al. [31] implemented BFV on FPGAs, emphasizing fast external memory access for high performance.

This work adopts GPU acceleration for its strong performance, programmability, and accessibility. GPUs' parallelism suits the data-parallel nature of FHE components (e.g., NTT) and are easier to program than FPGAs through mature CUDA toolchains. While we focus on the BGV scheme, the underlying optimization strategies naturally extend to other key-switching-based schemes such as BFV and CKKS. Our contribution distinguishes itself by accelerating BGV homomorphic multiplications, supporting larger ring dimensions, and integrating GPU acceleration directly into OpenFHE, minimizing development overhead compared to standalone GPU-based FHE implementations.

TABLE I
Execution time % of EvalMult as number of multiplications increases.

# of Multiplications	1	5	12	24
EvalMult Exec. Time %	12	27	50	69

IV. Acceleration Opportunities

To identify opportunities for performance improvement, we profiled OpenFHE with various computational depths for homomorphic multiplications using the CLion profiler based on *Perf* and *DTrace*. This focus on multiplication is motivated by prior work [32], which conducted a fine-grained

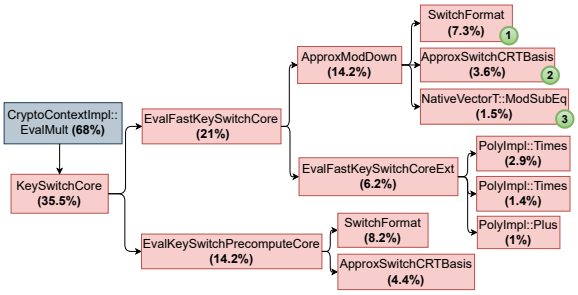

Fig. 1. BGV Homomorphic Multiplication Function Call Chain in OpenFHE.

performance analysis of BGV operations in OpenFHE and identified homomorphic multiplication as the most computationally expensive operation, significantly outweighing the cost of additions and other operations. The profiled FHE application encrypts two plaintexts, perform a specific number of consecutive homomorphic multiplications, and then decrypt the result. Table I shows the increasing impact of homomorphic multiplication (EvalMult) on the execution time as the number of consecutive multiplications grows (1, 5, 12, and 24). For a single multiplication, EvalMult accounts for over 12% of the total execution time, rising to approximately 70% with 24 multiplications, making it the dominant factor in end-to-end performance.

Furthermore, Figure 1 presents the function call tree of EvalMult along with the end-to-end execution time percentage of each function, using the CLion profiler on the profiled FHE application. Our analysis reveals that homomorphic multiplication primarily consists of two major components: i) Key Switching (`SchemeBase::KeySwitchCore`), and ii) Multiplication Core (`LeveledSHERNS::EvalMult`).

As shown in Figure 1, key switching dominates the homomorphic multiplication and is a rather computationally expensive operation as it accounts for 35.5% of total execution time. Furthermore, key switching involves polynomial multiplications and modular reductions, both of which get bigger as the ring dimension grows; hence they tend to become increasingly costly. Existing CPU-based implementations struggle to efficiently handle key switching for large ring dimensions due to limited parallelism.

Given the structured nature of key switching computations, there is an opportunity for GPUs to offer significant acceleration. Modern GPUs offer high-throughput arithmetic and efficient memory access patterns that can be leveraged to parallelize polynomial arithmetic and modular operations. By exploiting GPU acceleration, the performance of key switching can be significantly improved, making FHE more practical for real-world applications.

V. Design & Implementation

As noted in previous work [32], a common challenge in GPU acceleration is related with the data types compatibility, which in our case creates compatibility issues between OpenFHE's custom classes and

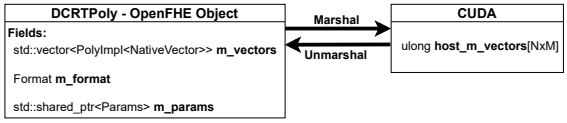

Fig. 2. Marshaling/Unmarshaling.

CUDA's primitive types. OpenFHE employs custom classes such as `DCRTPoly` for double-CRT polynomials and `NativeInteger`/`NativeVector` for polynomial coefficients, whereas CUDA primarily operates on primitive data types (e.g., `int`, `float`). To address this, we implemented a transparent transformation process that involves *marshaling and unmarshaling* (Figure 2). This process efficiently transforms OpenFHE objects into CUDA-compatible data structures before computation and converts them back afterward, ensuring seamless integration.

To accelerate FHE computations by parallelizing and offloading parts of the multiplication operation on GPUs, we followed two design principles:

1) **Achieve a positive compute-to-transfer ratio**, ensuring that computation benefits outweigh data transfer costs.
2) **Minimize data transfer overhead** by (i) caching reusable data on the GPU and (ii) overlapping data transfers with computations.

These principles are crucial, as prior work [32] has demonstrated that data transfers along with the data transformation of OpenFHE objects can be primary bottlenecks in end-to-end performance. Additionally, OpenFHE currently lacks a standardized interoperability interface for accelerator backends, necessitating every contributors to integrate with a custom interface for their GPU implementation. This absence introduces additional development challenges, as the complexity of the code base becomes higher and its maintainability gets lower. As a result, the scope of OpenFHE components that can be efficiently offloaded on an accelerator within a single research effort is inherently constrained.

A. Analysis of the Parallelizable ApproxModDown Function

As shown in Figure 1, key switching accounts for 35.5% of the total execution time, with its two main subcomponents being `EvalFastKeySwitchCore` (21.0%) and `EvalKeySwitchPrecomputeCore` (14.2%). Since `EvalFastKeySwitchCore` dominates the key switch operation, we prioritize optimizing its execution. Within `EvalFastKeySwitchCore`, `ApproxModDown` contributes 14.2%, making it the most computationally intensive function. This function implements an approximate modulus reduction algorithm, which is crucial for managing noise growth in homomorphic encryption. It consists of three key subroutines: `SwitchFormat` (which performs NTT transformations), `ApproxSwitchCRTBasis`, and `ModSubEq`. Given that `ApproxModDown` accounts for a substantial portion of execution time, includes highly parallelizable algorithms such as NTT transformations (in `SwitchFormat`), and features reusable components

like `SwitchFormat` and `ApproxSwitchCRTBasis`, it presents an ideal target for GPU acceleration.

a) SwitchFormat (Step 1): The first step in `ApproxModDown` applies an inverse Number Theoretic Transform (iNTT) to the polynomial input. This operation switches the polynomial's representation from the EVALUATION form back to the COEFFICIENT form. The NTT is a discrete transform that operates over a finite field, making it particularly suitable for modular arithmetic in cryptographic applications [33]. It enables efficient polynomial multiplication by converting convolution operations into pointwise multiplications, significantly reducing computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ [34].

b) ApproxSwitchCRTBasis (Step 2): In the next step, the `ApproxSwitchCRTBasis` function converts the polynomial from one modulus set to another. This transformation is fundamental in key switching and it enables compatibility between different modulus sets in Residue Number System (RNS) arithmetic. Specifically, it maps a polynomial defined over modulus set $Q = \{q_1, \ldots, q_i\}$ to a new modulus set $P = \{p_1, \ldots, p_k\}$. The transformation is efficiently computed using precomputed modular inverses and reductions, and it consists of modular multiplications, 128-bit multiplications, additions and Barrett reductions.

c) SwitchFormat (Step 3): The following step transforms the polynomial into the EVALUATION form for subsequent computations. This is achieved by invoking again the `SwitchFormat` function, which at this stage performs a forward NTT (fNTT) to return the polynomial to the EVALUATION representation.

d) ModSubEq (Step D): The final step in `ApproxModDown` applies an in-place modular subtraction to refine the output. This operation ensures that the polynomial coefficients remain within the appropriate range for continued homomorphic computations.

B. Parallelization of ApproxModDown

To fully utilize GPU resources, we redesigned the implementation of the ApproxModDown function in CUDA using the following techniques:

- **Breakdown and Composition of Pipeline:** We break down ApproxModDown into its sub-components (iNTT, ApproxSwitchCRTBasis, fNTT, and modular arithmetic), implementing each as a separate CUDA kernel.
- **Batch-Level Parallelism:** Instead of processing polynomials sequentially, we organize data into batches, thereby ensuring high GPU occupancy.

The computational dataset is structured as a two-dimensional array: One dimension corresponds to the modulus set (P or Q). The other dimension represents the polynomial's ring size.

However, ApproxModDown cannot be implemented as a single, continuous pipeline. This is due to ApproxSwitchCRTBasis, which transforms the polynomial from the smaller modulus set P to the larger modulus set Q, altering the dataset

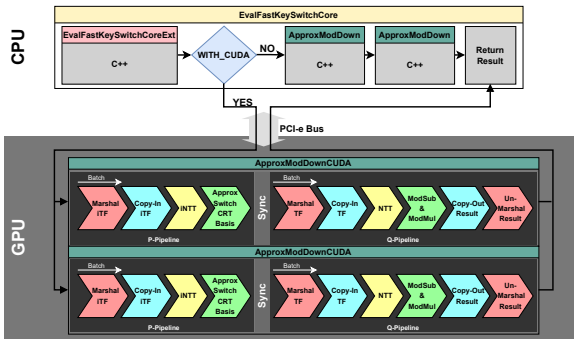

Fig. 3. ApproxModDownCUDA design & its interoperability with the OpenFHE library.

structure. To accommodate this, we split the execution into two distinct sub-pipelines:

1) **P-Pipeline:** Operates within modulus set P.
2) **Q-Pipeline:** Processes data in modulus set Q.

To enable efficient pipelined execution, we leverage asynchronous CUDA operations (e.g., `cudaMemcpyAsync`) and multiple CUDA streams. This approach provides several advantages:

- **Hiding data marshaling and transfer costs** by overlapping memory copies with computation.
- **Maximizing GPU utilization** by ensuring either data movement or computation is always active, reducing idle time.
- **Efficient batch scheduling**, allowing independent pipeline stages to progress while minimizing synchronization overhead.

As illustrated in Figure 3, our GPU-accelerated implementation integrates seamlessly within the OpenFHE execution flow. When GPU acceleration is enabled, ApproxModDownCUDA handles marshaling, data transfers, and the execution of the CUDA kernel in a streamlined process. Execution begins with marshaling necessary parameters (e.g., inverse Twiddle Factors for iNTT) and transferring them to the device. The **P-Pipeline** then computes the inverse Number Theoretic Transform (iNTT) on the input polynomial, followed by ApproxSwitchCRTBasis, which transforms the polynomial into modulus set Q. Once this step completes, the execution continues to the **Q-Pipeline**, where the forward NTT is performed, followed by modular arithmetic operations to finalize the result. Finally, the processed data is transferred back to the host and is unmarshaled into its original format.

C. Parallelization of EvalFastKeySwitchCore

As shown in Figure 3, the proposed design implements two instances of ApproxModDownCUDA which are executed asynchronously. The rationale behind this decision is that the ApproxModDown is invoked twice to process a single polynomial which is split into two `DCRTPoly` instances. Hence, we spawned two separate CPU threads, each invoking one instance of ApproxModDownCUDA with distinct data.

TABLE II
EXPERIMENTAL TESTBED.

Hardware	
Processor	Intel Core i9-13900K
P-Cores	8 (32 HyperThreads) @ 3 GHz
E-Cores	16 @ 2.2 GHz
RAM	64GB
GPU	NVIDIA GeForce RTX 4090 (Ada) Cores: 16384 Memory: 24 GB
Software	
Operating System	PopOS 22.04 LTS (Kernel 6.2.0-39-generic)
CUDA Driver	565.57.01 (CUDA 12.6)
OpenFHE	v1.0.3

TABLE III
CONFIGURATION PARAMETERS FOR OPENFHE AND THEIR CORRESPONDING GPU BLOCK AND THREAD ALLOCATIONS FOR VARIOUS COMPUTATIONAL DEPTHS.

Comput. Depth	Modulus	Cyclotomic	Ring Dimension	Blocks	Threads
1	65,537	16,384	8,192	8	1,024
5	65,537	32,768	16,384	16	1,024
12	65,537	65,536	32,768	32	1,024
24	786,433	131,072	65,536	64	1,024

This design decision improves the resource utilization of the GPU, and results in eliminating the idle time of the GPU while also maximizing throughput.

VI. EVALUATION

A. Experimental Methodology

To evaluate the performance of the parallel implementation, we conducted experiments on a testbed with both CPU and GPU, as detailed in Table II. All reported measurements are the average of one hundred executions. We used the std::chrono C++ library to obtain precise timing data for: a) end-to-end execution time (Enc. + Comp. + Dec.), b) homomorphic multiplications (Comp.) and, c) the ApproxModDown function, both on the CPU and offloaded to the GPU.

1) OpenFHE Configuration: Table III presents the configuration of the computational depth of the homomorphic multiplications, the modulus, the cyclotomic, and the ring dimensions for our experiments. Additionally, it reports the allocation of the GPU blocks and threads that we evaluated for each computational depth.

B. Performance Analysis

Figure 4 presents three performance metrics comparing the proposed GPU-based system to the baseline vanilla OpenFHE system running on CPU. The x-axis shows computational depth, with the blue line representing relative end-to-end performance. Additionally, we present two further lines that correspond to individual sub-parts of the end-to-end execution. The orange line corresponds to the performance of the homomorphic multiplication, and the green line shows the performance of the ApproxModDown function itself.

At low depths (1 to 5), GPU performance lags behind the baseline, with end-to-end execution 0.29x and 0.7x slower, and homomorphic multiplication time 0.53x (depth 1) and 0.97x (depth 5) lower. This reflects the overhead of offloading to the

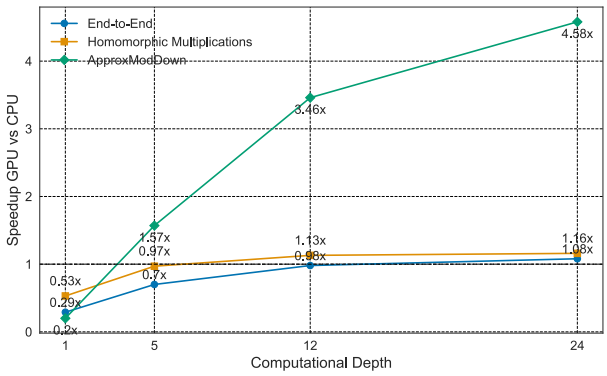

Fig. 4. Speedup of the ApproxModDown, the Homomorphic Multiplications, and the end-to-end accelerated implementation against the baseline.

GPU at lower workloads. However, as the number of consecutive multiplication increases, GPU acceleration becomes more effective. At a depth of 12, the GPU matches the CPU's end-to-end time (0.98×) and surpasses it in homomorphic operations (1.13× speedup). At a depth of 24, the GPU outperforms the baseline by 1.08× for end-to-end time and 1.16× for a single multiplication. A key factor in this improvement is the ApproxModDown function, which sees dramatic speedup at higher depths—3.46× at 12 multiplications and 4.58× at 24 multiplications. This indicates that the ApproxModDown function benefits considerably from GPU parallelization, thereby becoming a major contributor to the overall performance gains as the computational depth increases.

The 8% end-to-end execution time improvement at 24 multiplications is notable, considering only about 14% of the overall execution path (see figure 1) was offloaded to the GPU. This suggests that selectively accelerating key components—without reimplementing the entire library—can yield substantial gains, with room for further improvement by offloading additional parts. Moreover, this is expected to deliver even greater gains, as the copy-in and copy-out overhead has already been incurred.

VII. CONCLUSION

This work presented a novel approach to address the computational challenges of FHE associated with the BGV scheme. We provided an in-depth analysis of the computationally expensive components within OpenFHE for performing homomorphic operations and identified key switching, in particular, as a prime candidate for GPU acceleration. We detailed the design decisions made during the development of our parallel GPU implementation.

Our implementation achieved up to 4.58× speedup over vanilla OpenFHE ApproxModDown, with 1.16× improvement per homomorphic multiplication and an 1.08× reduction in end-to-end execution time for 24 consecutive multiplications, demonstrating the potential of GPU acceleration for practical FHE. In addition, this work novels in its specific focus on the BGV scheme while it reuses and seamlessly integrates with an open-source, and actively optimized FHE library, OpenFHE.

Future work includes offloading additional FHE components to GPUs leveraging existing data transfers and exploring FPGA-based acceleration strategies.

ACKNOWLEDGMENT

This work is supported by the European Union's Horizon Europe programme under grant agreement No 101070670 (ENCRYPT). In addition, this work is funded by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee (10039809).

REFERENCES

[1] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford, CA, USA, 2009, aAI3382729.

[2] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption," 2021.

[3] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, p. 34–91, Jan. 2020. [Online]. Available: https://doi.org/10.1007/s00145-019-09319-x

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 309–325. [Online]. Available: https://doi.org/10.1145/2090236.2090262

[5] A. A. Badawi, A. Alexandru, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, C. Pascoe, Y. Polyakov, I. Quah, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "OpenFHE: Open-Source Fully Homomorphic Encryption Library," Cryptology ePrint Archive, Paper 2022/915, 2022, https://eprint.iacr.org/2022/915. [Online]. Available: https://eprint.iacr.org/2022/915

[6] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[7] 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.

[8] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[9] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, 2011, pp. 97–106.

[10] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, Nov. 2013. [Online]. Available: https://doi.org/10.1145/2535925

[11] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Advances in Cryptology – EUROCRYPT 2012*, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 465–482.

[12] K. O. W. Group, "The OpenCL C Specification," Online: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html, Last Access: June 2024.

[13] NVIDIA, "NVIDIA CUDA Toolkit," Last Access: June 2024. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[14] oneAPI, "The oneAPI Programming Model," Last Access: June 2024. [Online]. Available: https://www.oneapi.io/

[15] J. Fumero, A. Stratikopoulos, and C. Kotselidis, *Heterogeneous Programming Models*. Cham: Springer International Publishing, 2024, pp. 37–56. [Online]. Available: https://doi.org/10.1007/978-3-031-49559-5_3

[16] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the Institute of Radio Engineers*, vol. 96, no. 5, pp. 879–899, May 2008.

[17] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Comput. Surv.*, 2002. [Online]. Available: https://doi.org/10.1145/508352.508353

[18] Zama, "Concrete: TFHE Compiler that converts python programs into FHE equivalent," 2022, https://github.com/zama-ai/concrete.

[19] C. Mouchet, J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: a Multiparty Homomorphic Encryption Library in Go," in *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography (WAHC)*. HomomorphicEncryption.org, 2020. [Online]. Available: https://homomorphicencryption.org/wp-content/uploads/2020/12/wahc20_demo_christian.pdf

[20] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-Performance FV Somewhat Homomorphic Encryption on GPUs: An Implementation using CUDA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, p. 70–95, May 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/875

[21] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 941–956, 2021.

[22] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using GPU," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 2800–2803.

[23] A. A. Badawi, B. Veeravalli, and K. M. Mi Aung, "Faster number theoretic transform on graphics processors for ring learning with errors based cryptography," in *2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, 2018, pp. 26–31.

[24] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022. [Online]. Available: https://doi.org/10.1007/s11227-021-03980-5

[25] A. Ş. Özcan and E. Savaş, "Heongpu: a gpu-based fully homomorphic encryption library 1.0," *Cryptology ePrint Archive*, 2024.

[26] E. R. Türkoğlu, A. Ş. Özcan, C. Ayduman, A. C. Mert, E. Öztürk, and E. Savaş, "An accelerated gpu library for homomorphic encryption operations of bfv scheme," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 1155–1159.

[27] J. Kim, W. Choi, and J. H. Ahn, "Cheddar: A swift fully homomorphic encryption library for cuda gpus," 2024. [Online]. Available: https://arxiv.org/abs/2407.13055

[28] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A cuda-accelerated word-wise homomorphic encryption library," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 5, pp. 4895–4906, 2024.

[29] R. Agrawal, L. Bu, and M. A. Kinsy, "Fast Arithmetic Hardware Library For RLWE-Based Homomorphic Encryption," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 206–206.

[30] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An Architecture for Computing on Encrypted Data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

[31] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, 2018.

[32] O. Papadakis, M. Papadimitriou, A. Stratikopoulos, M. Xekalaki, J. Fumero, N. Foutris, and C. Kotselidis, "Towards gpu accelerated fhe computations," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2024, pp. 694–699.

[33] D. Harvey, "Faster arithmetic for number-theoretic transforms," *Journal of Symbolic Computation*, vol. 60, pp. 113–119, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747717113001181

[34] A. Satriawan, R. Mareta, and H. Lee, "A complete beginner guide to the number theoretic transform (ntt)," Cryptology ePrint Archive, Report 2024/585, 2024, available at https://eprint.iacr.org/2024/585.pdf, last accessed: 9 March 2025.