BASALISC: Programmable Hardware Accelerator for BGV Fully Homomorphic Encryption

Distribution Statement A: Approved for Public Release, Distribution Unlimited

Robin Geelen[1*], Michiel Van Beirendonck[1*], Hilder V. L. Pereira[1], Brian Huffman[2], Tynan McAuley[3], Ben Selfridge[2], Daniel Wagner[2], Georgios Dimou[3], Ingrid Verbauwhede[1], Frederik Vercauteren[1] and David W. Archer[2]

[1] COSIC KU Leuven, Leuven, Belgium
firstname.lastname@esat.kuleuven.be
[2] Galois, Inc., Portland, OR, USA
huffman@galois.com
benselfridge@galois.com
dmwit@galois.com
dwa@galois.com
[3] Niobium Microsystems, Portland, OR, USA
firstname@niobiummicrosystems.com

Abstract. Fully Homomorphic Encryption (FHE) allows for secure computation on encrypted data. Unfortunately, huge memory size, computational cost and bandwidth requirements limit its practicality. We present BASALISC, an architecture family of hardware accelerators that aims to substantially accelerate FHE computations in the cloud. BASALISC is the first to implement the BGV scheme with fully-packed bootstrapping – the noise removal capability necessary for arbitrary-depth computation. It supports a customized version of bootstrapping that can be instantiated with hardware multipliers optimized for area and power.

BASALISC is a three-abstraction-layer RISC architecture, designed for a 1 GHz ASIC implementation and underway toward 150mm^2 die tape-out in a 12nm GF process. BASALISC's four-layer memory hierarchy includes a two-dimensional conflict-free inner memory layer that enables 32 Tb/s radix-256 NTT computations without pipeline stalls. Its conflict-resolution permutation hardware is generalized and re-used to compute BGV automorphisms without throughput penalty. BASALISC also has a custom multiply-accumulate unit to accelerate BGV key switching.

The BASALISC toolchain comprises a custom compiler and a joint performance and correctness simulator. To evaluate BASALISC, we study its physical realizability, emulate and formally verify its core functional units, and we study its performance on a set of benchmarks. Simulation results show a speedup of more than 5,000× over HElib – a popular software FHE library.

Keywords: Fully homomorphic encryption · Brakerski-Gentry-Vaikuntanathan · Hardware accelerator · Application-specific integrated circuit

1 Motivation

Fully Homomorphic Encryption (FHE) [RAD+78, Gen09, ACC+18] offers the promise of confidentiality-preserving computation over sensitive data in a variety of theoretical and practical applications, ranging from new cryptographic primitives to machine learning as

[*] R. Geelen and M. Van Beirendonck contributed equally to this research.

a service. Unfortunately, the utility of FHE is severely limited by its high memory size, memory bandwidth and high computational overhead. The typical result - computation that runs many orders of magnitude slower than insecure computation - prevents broad adoption. Although new schemes have markedly improved FHE performance [BGV14, CGGI20, BIP+22, CKKS17], and highly optimized FHE libraries [SEA22, HS14, Lat22, CJL+20, PRR17] are now available, FHE still remains orders of magnitude beyond acceptable performance limits for most potential applications.

In other computational domains where performance on general purpose processors is problematic, innovation has turned to purpose-built *accelerators*, tuned to exploit domain-specific characteristics of computation. DSP accelerators, arguably starting with the Texas Instruments TMS320 DSP family [LFS87] in 1983, are perhaps the first example of this approach. More recently, Graphics Processing Units (GPUs) have become popular for accelerating video stream processing and hash function computation. Our FHE accelerator, BASALISC, follows this approach in pursuit of bringing the throughput of FHE computation within an order of magnitude relative to cleartext computation [Ron20].

Contributions. We summarize the contributions of BASALISC as follows:

- BASALISC accelerates BGV arithmetic for a large range of parameters. In contrast to prior accelerators, BASALISC is the first to support and implement fully-packed BGV bootstrapping directly in hardware to enable unlimited-depth FHE computations. We propose a novel version of bootstrapping that is compatible with NTT-friendly primes. In contrast to prior work, BASALISC instantiates its multipliers exclusively to these NTT-friendly primes, which saves 46% logic area and 40% power consumption.
- BASALISC implements a massively parallel radix-256 NTT architecture, using a conflict-free layout, a corresponding layout permutation unit, and a twiddle factor generator. These units are deeply interleaved with the on-chip memory and provide a massive 32 Tb/s NTT throughput. In addition, we show that one can efficiently generalize the required layout permutation unit to compute BGV automorphisms without additional silicon area.
- BASALISC is a comprehensive RISC-like architecture with a three-level Instruction Set Architecture (ISA) that allows for reasoning at diverse levels of executive abstraction. It adopts a four-level memory hierarchy purpose-built to address common FHE memory bottlenecks, including a mid-level 64 MB on-chip Ciphertext Buffer (CTB). At the lowest level, a massively parallel multiply-accumulate unit with integrated 16-entry register file allows accelerating tight BGV key switching loops, asynchronously and independently of the CTB.
- BASALISC is placed and routed with 150mm^2 die size and 1 GHz operational frequency in a 12nm low-power Global Foundries process. Critical hardware logic is emulated and formally verified for correctness. BASALISC is evaluated on a set of micro- and macro-benchmarks, showing more than 5,000× speedup over HElib.

2 Preliminaries

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) provides a simple use model to securely outsource computation on sensitive data to a third party. Informally, the FHE model enables a user to encrypt their data m into a ciphertext $c = \mathrm{Enc}(m)$, then send it to a third party, who can compute on c. The third party produces another ciphertext c' encrypting $f(m)$ for some desired function f. This is done by representing the function in terms of the operations provided by the scheme, typically addition and multiplication, and computing these operations on the encrypted data. We say that f was computed homomorphically.

In FHE, the third party receives only ciphertexts and the public key, but never the secret key that allows decryption. As a result, the sensitive inputs are protected under the security of the encryption scheme. Because the result of the computation remains encrypted, the output also remains unknown to the third party: only the holder of the secret key can decrypt and access it. This scenario is illustrated in Figure 1.

To achieve security, the ciphertexts of all FHE schemes are noisy. This noise is added to the input data during encryption and removed during decryption. Each homomorphic operation, such as an addition or a multiplication, increases the noise in the resulting ciphertext. Decryption can still recover the correct result, provided that the noise is small enough. Therefore, we can compute only a limited number of homomorphic operations before we reach the limit of decryption failure.

Because multiplications increase ciphertext noise much more than additions, we usually model noise growth by the number of sequential multiplications. Computing the product $\prod_{i=1}^{L} m_i$ requires a *multiplicative depth* of $\lceil \log_2(L) \rceil$. This is accomplished by writing the product in a tree structure, with each leaf node representing one of the factors. Note that FHE suffers from a general trade-off between computational cost and tolerating a larger L: one can increase the encryption parameters as to obtain more multiplicative depth, but in doing so, the homomorphic operations become slower and the size of ciphertexts larger.

To support the computation of functions regardless of their multiplicative depth, FHE uses *bootstrapping*. This operation reduces noise by decrypting a ciphertext homomorphically. Unfortunately, bootstrapping is very expensive, so its use is often minimized. There are several techniques in the FHE literature to slow down the noise growth, and thus delay bootstrapping. This work employs *key switching* and *modulus switching* [BGV14]. In practice, bootstrapping and key switching tend to heavily dominate computation and data movement costs of an application: in a simple 1,024-point, 10-feature logistic regression, we see that these tasks account for over 95% of the computational effort and the vast majority of data movement.

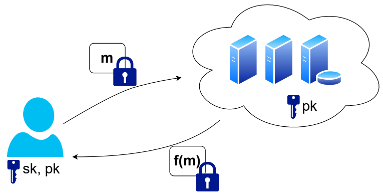

Figure 1: FHE used in a typical commercial application.

In summary, the implementation challenges of FHE are the high complexity of computation, the large ciphertext expansion ratio (large polynomials with integer coefficients of a 1000 bits or more), and the proportion of effort needed in bootstrapping (or delaying it) in sufficiently complex programs. In the remainder of this paper, we examine the magnitude of these challenges and how they impact the design of our FHE accelerator.

2.2 The BGV Cryptosystem

BASALISC targets the homomorphic encryption scheme BGV [BGV14]. Plaintexts and ciphertexts are represented by elements in the ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ with N a power of 2. Those elements are thus polynomials reduced modulo $X^N + 1$, which is implicit in our notation. BGV guarantees finite data structures by additionally reducing the coefficients: the plaintext space is computed modulo t (denoted $\mathcal{R}_t$), and the ciphertext space is a pair of elements modulo q (denoted $\mathcal{R}_q^2$). Reduction modulo m (with $m = t$ or q) is explicitly denoted by $[\cdot]_m$ and is done symmetrically around 0 (i.e., in the set $[-m/2, m/2) \cap \mathbb{Z}$).

As with traditional ciphers, BGV has encryption and decryption procedures to move between the *plaintext space* and the *ciphertext space*. These operations are never executed by the server that performs the outsourced computation, and are therefore not implemented by BASALISC. However, the ciphertext format remains an important element of the BGV scheme and is essential to explain the homomorphic operations. A BGV ciphertext

$(\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_q^2$ is said to encrypt plaintext $\boldsymbol{m} \in \mathcal{R}_t$ under secret key $\boldsymbol{s}$ (which has small coefficients) if

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \boldsymbol{m} + t\boldsymbol{e} \pmod{q} \tag{1}$$

for some element $\boldsymbol{e}$ that also has small coefficients. The term $\boldsymbol{e}$ is called the *noise*, and it determines if decryption returns the correct plaintext: if $\boldsymbol{e}$ has coefficients roughly less than $q/2t$, then $\boldsymbol{m} + t\boldsymbol{e}$ does not overflow modulo q. We can therefore recover the plaintext uniquely as $\boldsymbol{m} = [[\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}]_q]_t$.

2.2.1 Basic Homomorphic Operations

Smart and Vercauteren [SV14] observed that for $t = p^r$ with p an odd prime, the plaintext space $\mathcal{R}_t$ is equivalent to $\mathbb{Z}_t^\ell$ for some ℓ that divides N. This technique - referred to as *packing* - allows us to encode ℓ numbers into one plaintext simultaneously. Addition and multiplication over tuples in $\mathbb{Z}_t^\ell$ are then performed entry-wise. As a result, one ciphertext can encrypt and operate on an entire tuple, which leads to significant performance benefits and memory reductions in practice.

When BGV is used in conjunction with packing, one can define three basic homomorphic operations. Let $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ and $(\boldsymbol{c}_0', \boldsymbol{c}_1')$ be two distinct ciphertexts that encrypt respectively the tuples $(m_1, ..., m_\ell)$ and $(m_1', ..., m_\ell')$, then we can perform the following operations:

- **Addition:** compute $\mathsf{ct}_{\text{add}} = ([\boldsymbol{c}_0 + \boldsymbol{c}_0']_q, [\boldsymbol{c}_1 + \boldsymbol{c}_1']_q)$. The underlying plaintext of ct_{add} is then $(m_1 + m_1', ..., m_\ell + m_\ell')$.
- **Multiplication:** compute $\mathsf{ct}_{\text{mul}} = ([\boldsymbol{c}_0 \cdot \boldsymbol{c}_0']_q, [\boldsymbol{c}_0 \cdot \boldsymbol{c}_1' + \boldsymbol{c}_1 \cdot \boldsymbol{c}_0']_q, [\boldsymbol{c}_1 \cdot \boldsymbol{c}_1']_q)$. Observe that the resulting ciphertext consists of three ring elements, but this can be reduced back to two with a post-processing step known as *key switching.* The underlying plaintext of ct_{mul} is then $(m_1 \cdot m_1', ..., m_\ell \cdot m_\ell')$.
- **Permutation:** compute $\mathsf{ct}_{\text{per}} = (\phi_k(\boldsymbol{c}_0), \phi_k(\boldsymbol{c}_1))$. The map ϕ_k (called *automorphism*) is parameterized by an odd integer k and defined as $\phi_k \colon c(X) \mapsto c(X^k)$. Gentry et al. [GHS12a] have shown that automorphisms induce a permutation on the elements of the encoded tuple. The underlying plaintext of ct_{per} is therefore a permutation of $(m_1, ..., m_\ell)$. Although the resulting ciphertext has only two elements, we still need post-processing by means of key switching.

The validity of these three operations can be verified by observing their effect on Equation 1. We refer to Zucca [Zuc18] for more details, including an analysis of the noise growth of each homomorphic operation.

2.2.2 Auxiliary Homomorphic Operations

Basic homomorphic operations lead to ciphertext expansion and noise growth. A product ciphertext, for example, consists of three elements and is encrypted under $(\boldsymbol{s}, \boldsymbol{s}^2)$ instead of $\boldsymbol{s}$. Moreover, the noise term is equal to $t\boldsymbol{e} \cdot \boldsymbol{e}'$, where $\boldsymbol{e}$ and $\boldsymbol{e}'$ are the noise terms of the input ciphertexts. A similar problem occurs during permutation, where the automorphism ϕ_k not only acts on the ciphertext and plaintext, but also on the secret key. The resulting ciphertext will therefore be encrypted under $\phi_k(\boldsymbol{s})$ instead of $\boldsymbol{s}$.

To prevent ciphertext expansion, maintain the same encryption key for each ciphertext, and slow down noise growth, BGV defines two auxiliary procedures:

- **Modulus switching:** given a ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_q^2$ and a modulus q', compute a new ciphertext $(\boldsymbol{c}_0', \boldsymbol{c}_1') \in \mathcal{R}_{q'}^2$ that decrypts with respect to q' instead of q. Modulus switching has the positive side effect of reducing the noise by a factor of q'/q.

- **Key switching:** given a key switching matrix $(\overrightarrow{\boldsymbol{k}_0}, \overrightarrow{\boldsymbol{k}_1})$ and either a product ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathcal{R}_q^3$ or a permuted ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_q^2$, compute a new ciphertext $(\boldsymbol{c}_0', \boldsymbol{c}_1') \in \mathcal{R}_q^2$ that decrypts directly under the secret key $\boldsymbol{s}$ using Equation 1. Thus key switching brings the ciphertext back to its original format.

Modulus switching is typically done just before multiplication in order to reduce the noise to its minimum level. Key switching is performed after multiplication or permutation to keep the ciphertext format consistent. We again refer to Zucca [Zuc18] for more details, including an analysis of the noise growth of both operations.

2.2.3 Bootstrapping

When the entire noise budget of a ciphertext is consumed (equivalently, when the modulus is depleted to its minimum value by successive modulus switchings), further homomorphic operations are no longer immediately possible. We can overcome this problem by means of a *bootstrapping* procedure that reduces the noise back to a lower level [Gen09]. Bootstrapping refreshes a ciphertext by running decryption *homomorphically*. One evaluates an adapted version of Equation 1, followed by a homomorphic rounding procedure. The state-of-the-art bootstrapping technique for BGV is implemented in the HElib software library [HS21].

3 Data Representation and Algorithms

Basic homomorphic operations can be implemented via arithmetic in $\mathcal{R}_q$, i.e., based on polynomial addition, multiplication and automorphism. In order to obtain efficient arithmetic in $\mathcal{R}_q$, two common tricks have been developed [GHS12b], and BASALISC employs them as well. Firstly, Section 3.1 explains how computations modulo q can be split into many smaller moduli q_i, based on the Chinese Remainder Theorem (CRT). This data representation is called a Residue Number System (RNS). Secondly, Section 3.2 explains conversion between the polynomial and frequency domain via the Number-Theoretic Transform (NTT), which is necessary for efficient multiplication and automorphism. Similarly to RNS, the NTT can also be interpreted in terms of the Chinese Remainder Theorem. Hence, the combination of using RNS for fast arithmetic modulo q and the NTT for fast polynomial arithmetic modulo $X^N + 1$, is referred to as *Double-CRT* representation.

3.1 Residue Number System

Suppose that the ciphertext modulus is given by the product $q = q_1 \cdot \ldots \cdot q_k$ of pairwise coprime numbers. Then computations in $\mathcal{R}_q$ can be reduced to simultaneous computations in the smaller rings $\mathcal{R}_{q_i}$ by applying the Chinese Remainder Theorem. This data representation is called a Residue Number System (RNS), and brings an asymptotic speedup factor of $\mathcal{O}(k)$. Moreover, it simplifies architecture design, because the size of each q_i is much smaller than q (a typical value is 32 bits for q_i versus more than 1000 bits for q).

3.2 Number-Theoretic Transform

In order to perform efficient polynomial multiplication in time $\mathcal{O}(N \log(N))$, we resort to the Number-Theoretic Transform (NTT). The NTT is a generalization of the Fast Fourier Transform (FFT) to finite fields, and allows us to use exact integer arithmetic, preventing round-off errors typical of real-valued FFT computations. Similar to the FFT, the NTT can be computed with the Cooley-Tukey algorithm that recursively re-expresses an NTT of size $N = N_1 N_2$ as N_2 inner NTTs of size N_1, followed by N_1 outer NTTs of size N_2. Before the outer NTT, each output of the inner NTT is multiplied by a twiddle factor:

$$X[k_1{+}N_1k_2] = \sum_{n_2=0}^{N_2-1} \left(\sum_{n_1=0}^{N_1-1} x[N_2n_1{+}n_2]\omega_{N_1}^{n_1k_1} \right) \omega_N^{n_2k_1}\omega_{N_2}^{n_2k_2}. \tag{2}$$

By choosing $N_1 = 2$ and $N_2 = N/2$ at each recursive decomposition or vice-versa, the well-known radix-two Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF) algorithms are obtained, respectively.

The NTT can be used directly for fast cyclic convolutions (polynomial multiplication modulo $X^N - 1$). However, BGV performs polynomial multiplication modulo $X^N + 1$, requiring *negacyclic* convolutions. Those can still be implemented with a regular NTT, but require an additional pre-multiplication of the input polynomials and post-multiplication of the output polynomial by an extra set of twiddle factors [AHU74].

3.3 Implemented Algorithms and Parameter Sets

BASALISC accelerates the five basic and auxiliary homomorphic operations. All of these can be built from three building blocks on vector operands: entry-wise addition and multiplication; entry permutation; and the NTT. For example, addition is coefficient-wise and handled directly in either the polynomial or frequency domain. On the other hand, multiplication and automorphism can only be handled in the frequency domain, respectively via entry-wise multiplication and entry permutation. Modulus and key switching are more complicated, and they need conversion between the polynomial and frequency domain via the NTT.

3.3.1 Supported Parameter Sets

As opposed to software implementations, hardware accelerators gain throughput benefits by supporting a limited range of commonly used parameters. We start with the realization that at least 128-bit security must be supported if BASALISC is to be interesting to real-world users. Based on this observation, we choose a parameter range that allows for an efficient implementation, while still retaining sufficient freedom for application design. A typical range for the ring dimension N, offering sufficient flexibility, is between 2^{14} and 2^{16}. BASALISC settles on a maximum value of $N = 2^{16}$, which allows ciphertext moduli up to $q = 2^{1782}$ at 128-bit security level. This gives a large number of multiplicative levels, even at a high-precision plaintext space (e.g., 31 levels at plaintext modulus $t = 127^3$ without bootstrapping; with bootstrapping, we get an arbitrary number of levels).

Table 1 shows the full parameter range supported by BASALISC and an example parameter set for illustration. The largest ciphertext modulus that appears during the basic homomorphic operations is denoted by Q; during key switching, however, the modulus is temporarily extended to QP. Concretely, our largest supported modulus is $QP \approx 2^{1782}$.

Table 1: BASALISC parameter ranges and examples.

Parameter	Range	Example
Security parameter	N/A	128 bits
Ring dimension N	512 – 65536	65536
Plaintext modulus t	> 2	127^3
Ciphertext packing ℓ	2 – 65536	64 slots
Max $\log_2(QP)$ for key switching	20 – 1782	1782 bits
Max $\log_2(Q)$ for ciphertext	20 – 1782	1263 bits
Max multiplicative depth L	N/A	31

3.3.2 Algorithmic Details

Currently, the term BGV refers to a family of related algorithms that are derived from the original scheme [BGV14]. One assumption of the original scheme is that the factors of the ciphertext modulus satisfy $q_i = 1 \pmod{t}$. This can be interpreted as a $\log_2(t)$-bit restriction on q_i, which limits the freedom in the selection of the ciphertext modulus and poses a lower bound on the factors of the RNS chain. HElib implements an improved variant of the BGV scheme that does not require this restriction [HS20]. However, this improved version needs to keep track of a *correction factor* $\kappa \in \mathbb{Z}_t^*$ and encrypts $\kappa \cdot \boldsymbol{m}$ instead of $\boldsymbol{m}$. Each ciphertext is tagged with such a correction factor, which is removed upon decryption by multiplying with κ^{-1}. BASALISC implements the HElib version of BGV. Correction factor management is done by our compiler Artemidorus (see Section 8).

Due to algebraic constraints, the NTT is only defined for prime moduli of the shape $q_i = 1 \pmod{2N}$. These special moduli are referred to as *NTT-friendly primes*. In the case of our ring dimension $N = 2^{16}$, this puts a lower bound of 17 bits on the size of q_i. Coupled with the requirement to have a sufficient amount of moduli available to reach $\log_2(QP) = 1782$ bits, a simple analysis can show that we need q_i of at least 26 bits. In practice, however, BASALISC employs 32-bit moduli, because it gives a better utilization for the on-chip memory buffer and simplified interaction with the external memory. Furthermore, both 26-bit and 32-bit moduli result in the same complement of arithmetic units within our silicon area budget. For the example parameter set of Table 1, Q is a product of 42 primes and P is a product of 14 additional primes.

4 NTT-Friendly Bootstrapping

Since the number-theoretic transform is only defined for NTT-friendly primes, we restrict BASALISC's hardware multipliers to these NTT-friendly primes (see Section 7.4 for the motivation of this design choice). However, all current bootstrapping implementations switch to a specially-shaped modulus [HS21, CH18], either $q = p^e + 1$ or $q = p^e$, which is not an NTT-friendly prime in general. For software implementations, this is not an issue, because CPUs provide a general-purpose instruction set. However, with our specialized hardware multipliers, bootstrapping cannot be implemented directly.

We propose a generalized version of bootstrapping that works with NTT-friendly primes exclusively, and can still be evaluated with the same computational cost as regular bootstrapping. It relies on a simplified decryption lemma that was proposed earlier [GV22]. Our contribution is to implement this lemma in BASALISC, which requires non-trivial RNS operations as explained in the remainder of this section.

Lemma 1 (Simplified decryption [GV22])**.** *Let p be a prime number, and let $e > r \geqslant 1$ and q be sufficiently high parameters with $\gcd(q, p) = 1$. If $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ is a BGV encryption of $\boldsymbol{m}$ under plaintext modulus p^r and ciphertext modulus q, then it can be decrypted as*

$$\boldsymbol{c}_i' \leftarrow [p^{e-r}\boldsymbol{c}_i]_q, \quad \boldsymbol{w} \leftarrow [q^{-1} \cdot (\boldsymbol{c}_0' + \boldsymbol{c}_1' \cdot \boldsymbol{s})]_{p^e} \quad \textit{and} \quad \boldsymbol{m} \leftarrow [q \cdot \lfloor \boldsymbol{w}/p^{e-r} \rceil]_{p^r},$$

where $q^{-1} \cdot q = 1 \pmod{p^e}$, and $\lfloor \cdot \rceil$ denotes coefficient-wise rounding to the nearest integer.

Our NTT-friendly bootstrapping evaluates Lemma 1 in the homomorphic domain, and it uses two primitives from Bajard et al. [BEHZ16]. Consider two coprime moduli

$$Q = \prod_{i=1}^{k} q_i \qquad \text{and} \qquad P = \prod_{j=1}^{\ell} p_j.$$

The first primitive - *fast base extension* - extends the modulus from Q to QP and is defined as follows. Given an element $\boldsymbol{a} \in \mathcal{R}_Q$ in polynomial representation, compute

$$\textsc{FastBaseExt}_{Q\to QP}(\boldsymbol{a}) = \left(\left[\sum_{i=1}^{k}\left[\boldsymbol{a}\cdot\left(\frac{Q}{q_i}\right)^{-1}\right]_{q_i}\cdot\frac{Q}{q_i}\right]_{p_j}\right)_{1\leqslant j\leqslant \ell} \in \mathcal{R}_{QP}. \tag{3}$$

Fast base extension possibly incurs additional overflows modulo Q. This means that the output will be equal to $\boldsymbol{a} + Q\boldsymbol{r}$ for some $\boldsymbol{r} \in \mathcal{R}$. However, denoting the infinity norm by $||\cdot||_\infty$, the overflows will be upper bounded as $||\boldsymbol{r}||_\infty \leqslant k/2$.

The second primitive - *small Montgomery reduction* - takes an element $\boldsymbol{b} \in \mathcal{R}_{QP}$ in polynomial representation subject to $||\boldsymbol{b}||_\infty \ll P\cdot m$, and it outputs $\boldsymbol{c} \in \mathcal{R}_Q$ such that $\boldsymbol{c} = \boldsymbol{b}\cdot P^{-1} \pmod{m}$. Moreover, the coefficients of the result will be reduced modulo m, i.e., they are upper bounded as $||\boldsymbol{c}||_\infty \leqslant (1+\epsilon)m/2$ for some $\epsilon \ll 1$. This functionality is denoted by $\textsc{SmallMont}_{QP\to Q}(\boldsymbol{b}, m)$. During bootstrapping, we use small Montgomery reduction to compensate for the overflows of fast base extension and to compute reduction modulo p^e as required in the decryption formula.

NTT-friendly bootstrapping is specified in Algorithm 1 and is a translation of Lemma 1 to the homomorphic domain. We use three pairwise coprime moduli Q, q and b, where the first two are products of NTT-friendly primes, and the last one is an NTT-friendly prime. Conversion between a ciphertext and its ring elements is implicit in our notation: at every place in the algorithm, we have $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ with correction factor κ (and similarly for other ciphertexts). Note that the input ciphertext and all other variables are assumed to be in polynomial representation.

The algorithm first multiplies the ciphertext by p^{e-r} and converts to Montgomery representation with respect to the auxiliary base b. Then the modulus is lifted to $Q\cdot q\cdot b$, which causes extra overflows modulo q. These are compensated almost perfectly by the small Montgomery reduction on the next line, which also converts back from Montgomery to regular representation (i.e., it removes the additional b-factor). Then we reduce modulo p^e via another small Montgomery reduction, and as a side effect, the ciphertext gets multiplied by q^{-1}, which is required in the decryption formula. The resulting ciphertext $\mathsf{ct}'' = (\boldsymbol{c}_0'', \boldsymbol{c}_1'')$ is an encryption of $\boldsymbol{w}$ under plaintext modulus p^e and ciphertext modulus Q. Observe that taking the product with the secret key $\boldsymbol{s}$ is implicit and does not require any computation during bootstrapping [ASP13]. Finally, we perform the homomorphic rounding procedure in the same way as HElib and restore the original correction factor. The correction factor is also augmented with an additional factor of q^{-1}, which represents the multiplication by q that is required in the decryption formula. Note that our algorithm does not change key generation nor encryption, so it does not influence the security of BGV.

Algorithm 1 NTT-friendly bootstrapping

Input: $\mathsf{ct} \in \mathcal{R}_q^2$ with noise $\boldsymbol{e}$

Output: $\mathsf{ct}'' \in \mathcal{R}_{q''}^2$ with noise $\boldsymbol{e}''$ s.t. $\mathrm{Dec}(\mathsf{ct}'') = \mathrm{Dec}(\mathsf{ct})$ and $||\boldsymbol{e}''||_\infty/q'' \ll ||\boldsymbol{e}||_\infty/q$

```
function Bootstrap(ct)                                   ▷ Store correction factor $\kappa$ of ct
    for $i \in \{0,1\}$ do
        $\boldsymbol{c}_i' \leftarrow [\boldsymbol{c}_i \cdot p^{e-r} \cdot b]_q$                           ▷ Residues defined mod $q$
        $\boldsymbol{c}_i' \leftarrow \textsc{FastBaseExt}_{q\to Q\cdot q\cdot b}(\boldsymbol{c}_i')$          ▷ Residues defined mod $Q\cdot q\cdot b$
        $\boldsymbol{c}_i' \leftarrow \textsc{SmallMont}_{Q\cdot q\cdot b\to Q\cdot q}(\boldsymbol{c}_i', q)$       ▷ Reduce mod $q$ and drop $b$
        $\boldsymbol{c}_i'' \leftarrow \textsc{SmallMont}_{Q\cdot q\to Q}(\boldsymbol{c}_i', p^e)$          ▷ Reduce mod $p^e$ and drop $q$
    end for
    $\mathsf{ct}'' \leftarrow \lfloor \mathsf{ct}''/p^{e-r} \rceil$                  ▷ Same as in HElib with initial $\kappa'' \leftarrow 1$
    return $\mathsf{ct}''$                        ▷ Update correction factor as $\kappa'' \leftarrow \kappa'' \cdot \kappa \cdot q^{-1}$
end function
```

Table 2: Example opcodes (left) and micro-level operand addressing modes (right).

ISA	Opcode	Semantics
Macro	LOAD	Move data from distant to near memory
	KSW	Key switch a ciphertext
	MORPH	Perform automorphism on a ciphertext
Mid	MULI	Multiply a residue polynomial by constant
	NTT	Compute NTT of residue polynomial
	FBE	Fast Base Extension
Micro	NTT1	Perform iteration of first NTT pass
	MAC	Multiply operands and add to accumulator

Mode	Definition
$XXX	address in distant memory only for LOAD/STORE
rXXX	address in middle memory
tXX	register in near memory
nXXX	immediate 32-bit scalar
iXXX	index in moduli table

5 BASALISC Instruction Set Architecture

BASALISC is an adapted Reduced Instruction Set Computer (RISC) architecture with a three-level instruction set. This multi-level approach allows for reasoning at diverse levels of abstraction, and aids in assuring correctness of our system. Having a hierarchy of multiple intermediate representations and instruction sets, each with well-defined semantics, means that we can implement and test each stage of the compiler toolchain separately. In addition, different instruction set abstractions allow programmers to work at a higher level of abstraction while allowing compiler writers and library authors to reason about lower-level details such as scheduling and optimizations easily. For example, when writing a program to run on BASALISC, the programmer need not know about low-level data representations. We generate and reason over three distinct levels of instruction and typesystem abstraction:

- **Macro-instructions** are at the highest level, with the largest data types and the most complex operations. Entire ciphertexts, plaintexts, and key switching matrices are treated as basic data types at this abstraction level. Operations include ciphertext addition, multiplication, modulus and key switching, automorphisms, and bootstrapping. Details about data representation and algorithms that implement those operations are opaque at this level of abstraction.

- **Mid-level instructions** expose the Double-CRT data representation. The basic data type at this level is a residue polynomial (a polynomial in RNS representation) comprising up to 2^{16} 32-bit polynomial coefficients. Basic operations on these data types include pointwise modular addition and multiplication on vectors of coefficients; automorphisms; NTTs; and multiply-accumulate iterations commonly used in key switching. Also included in this list are memory management instructions that Load and Store data to and from off-chip memory.

- **Micro-instructions** correspond very closely with the specific operations performed by the processing elements (PEs). The basic data type at this level contains as many coefficient words (1024 or 2048) as can be processed simultaneously by a PE or accessed in one on-chip memory cycle. Instructions at this level are delivered via the Peripheral Component Interconnect Express (PCIe) interface to the BASALISC processor for execution. This instruction level also includes rudimentary machine control instructions.

The left subtable of Table 2 shows operation code mnemonics (opcodes) from each level of our instruction set. Together with opcodes, BASALISC uses operand specifiers (omitted from the table) with register-like addressing modes for all levels in its ISA and memory hierarchy. The right subtable of Table 2 shows addressing mode examples for operand specifiers at the Micro-instruction level. At this level, operands are either “chunks” of 2048 32-bit coefficients within a residue polynomial, 32-bit scalar values, or natural number indices into tables of moduli.

6 BASALISC Hardware Design

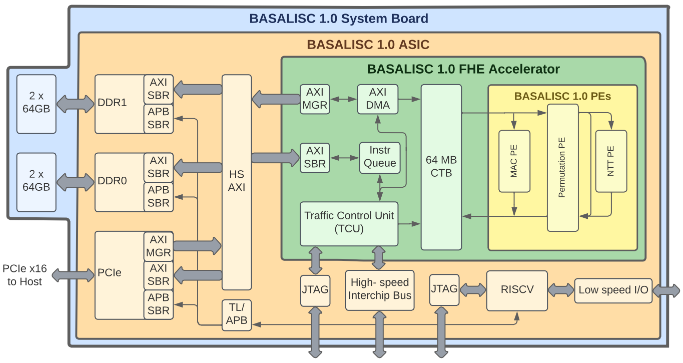

Figure 2: BASALISC System Diagram.

Figure 2 shows a system diagram of BASALISC. BASALISC is a single-chip FHE coprocessor, designed in a 12nm Global Foundries process, with additional off-chip memory; high-speed connectivity to its host system; and extensibility via a high-speed inter-chip interconnect. For its implementation, BASALISC employs a mature asynchronous logic process controlled by standard handshaking protocols [DSL$^+$18], allowing units to accelerate to a higher clock frequency whenever input data is available. Section 7.3 describes how this design choice helps to accelerate costly key switchings.

6.1 Memory Architecture

BASALISC includes four layers of memory hierarchy that exhibit diverse latencies and capacities. Table 3 depicts these four layers, where - typical of computer memory hierarchies - lower latency layers have smaller capacities. From farthest to nearest to the PEs, these four layers comprise off-chip distant memory (DRAM), a middle memory Ciphertext Buffer (CTB), a local Register File (RF) and Accumulator register (ACC) within the MAC PE. A significant difference between typical memory hierarchies and that of BASALISC is the working set size that each layer can hold. Still, we expect capacity limits of layers in our memory hierarchy to be a major limiter of system performance. In particular, we expect minimal locality of reference for key switching matrices, each of which is larger than the entire CTB. We now describe the DRAM array and the CTB, and defer the description of the MAC memory architecture to Section 7.3.

- **The 64 MB CTB** contains 2^{24} locations, each of which holds a 32-bit residue polynomial coefficient. In our largest supported parameter set, a single residue polynomial consists of $N = 2^{16} = 64\text{K}$ coefficients and occupies one entire page of the CTB. The CTB is a single-port SRAM array that can either read or write 2048 32-bit residue polynomial coefficients every machine cycle, providing a total bandwidth of 8 Tb/s (at 1 GHz operation) to our set of data Processing Elements (PEs) shown in yellow. As an advantage of FHE determinism, allocation and size of all data and operands are bound at compile-time. This allows the BASALISC CTB to be structured as an addressable set of ciphertext registers, instead of requiring the complex functionality of a run-time cache memory. This set of registers is compiler-managed with a true Least-Recently Used (LRU) replacement policy. CTB

Table 3: Memory hierarchy for ciphertext and key storage.

Memory	Capacity	Round-trip latency
Off-chip DRAM	256 GB	> 100 ns
CTB	64 MB	∼3 ns
MAC RF	128 kB	∼1.25 ns
MAC ACC	8 kB	0.625 ns

bandwidth is not materially affected by concurrent transfer between distant memory and the CTB: roughly at most 0.3% of CTB access cycles are used by our total distant memory bandwidth.

- **The 256 GB DRAM** serves as the staging area for data that is scheduled for processing and for results that are ready for retrieval by the host computer. In addition, for many applications, the 64 MB CTB is too small to hold the sizeable working sets of ciphertexts and key switching matrices. The DRAM array ensures that CTB capacity misses do not have to spill to host memory.

6.2 System Design

In contrast to other FHE hardware accelerators [SFK+21], BASALISC reduces cost and manufacturing risk by relying only on commercially available standard packaging, DRAM, and PCIe technologies and fits within 150mm^2 [Ron20]. From the top down, the BASALISC system can be described in different levels of hierarchy (see Figure 2):

- **(Blue) The BASALISC System Board** instantiates the distant DRAM memory using two DDR4 subsystems, each providing up to 128 GB of DRAM and 25.6 GB/s of bandwidth. At bottom left of the diagram is the 26 GB/s (near-peak) PCIe x16 channel that connects BASALISC to its host and carries data and instructions. We expect applications with a large working set to be performance-limited by our DDR4 bandwidth. The twin DDR4 interfaces allow us to maximize the practical throughput of the DRAM subsystem, by avoiding collisions between the PCIe-to-DRAM and FHE-to-DRAM access streams.
- **(Orange) The BASALISC ASIC** includes JTAG I/O for testing and debuging, a RISC-V CPU to configure BASALISC at startup, and the controllers and physical interfaces (PHYs) for DDR4 and PCIe. These PHYs connect to the 512-bit wide Advanced eXtensible Interface 4 (AXI4) interconnect that transfers data between the DDR, PCIe, and the CTB. Both the AXI4 and CTB operate at a target cycle time of 1 GHz. As a result, the AXI has a peak bandwidth of 32 GB/s for each endpoint connection, all running in parallel.
- **(Green) The BASALISC FHE Core Processor** includes the CTB, AXI4 infrastructure, and a Traffic Control Unit (TCU). The TCU maintains a batch-queue instruction buffer to manage instruction execution in the system. FHE programs are deterministic: the compiler knows in advance about the flow of instructions and data in memory. This allows the queue to be entirely compiler-managed and fairly short (128 to 1024 instructions, depending on our performance analysis).

7 BASALISC Processing Elements

BASALISC's on-chip PEs and their connection to the CTB are shown on the far right in Figure 2. The BASALISC PEs that rely on the CTB for data are the Multiply-Accumulate (MAC) PE (used in ciphertext addition, multiplication, and for kernels of operations such as key switching); the Permutation PE (used to permute data into preferred orders to achieve NTT processing, and also used for automorphisms); and the NTT PE (used to accomplish number-theoretic transforms efficiently). We describe their capabilities below.

Whereas Figure 2 shows single PE instances, their implementation is a massively multicore architecture that exploits innate parallelism in FHE ciphertext computations. FHE arithmetic in RNS representation offers four types of parallelism: *(i)* over multiple ciphertexts, *(ii)* over the polynomials within a ciphertext, *(iii)* over the residue levels of a polynomial, and *(iv)* over the coefficients of a residue polynomial. Prior work has

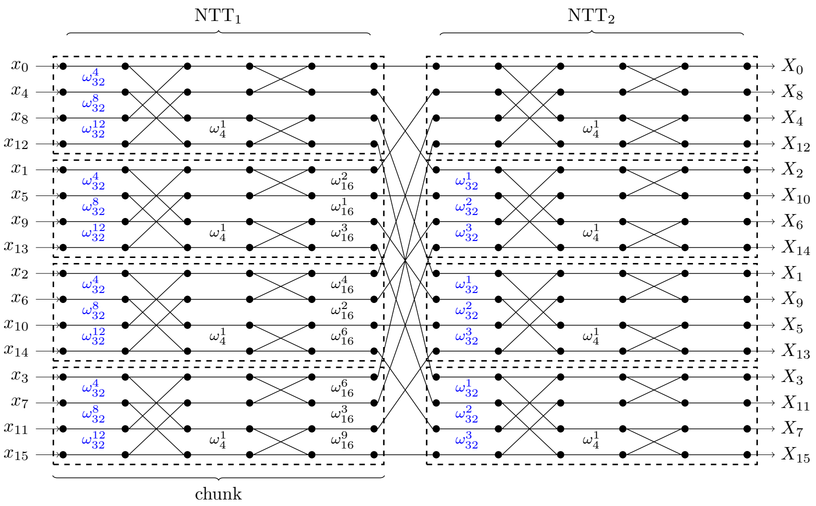

Figure 3: 16-point radix-4 negacyclic NTT flow graph. Extra negacyclic twiddles (in blue) are decomposed into two pre-multiply passes.

focused on *(iii)*, instantiating multiple so-called Residue Polynomial Arithmetic Units (RPAUs) [TRV20, RMA+21, MAK+22]. In contrast, BASALISC focuses on exploiting *(iv)*, due to two key observations. First, the number of residues decreases with the modulus level in the BGV scheme, leading to would-be idle RPAUs as the computation gets closer to bootstrapping. Second, as the lowest level of parallelism, coefficient-level parallelism offers the best opportunity to exploit locality of reference.

7.1 Number-Theoretic Transform PE

Because of the focus on coefficient-level parallelism, BASALISC implements a high-radix NTT PE. We expect that many BASALISC FHE applications will employ ring dimension $N = 65536 = 256^2$ to enable bootstrapping and thus arbitrary-depth computation. Thus, our NTT PE employs a radix-256 butterfly, allowing us to compute 65536-point NTTs with only two round trips to memory for each coefficient. NTTs of smaller sizes can be computed through shortcut paths in our NTT butterfly network.

Following the generalized Cooley-Tukey NTT description of Equation 2, a radix-256 NTT chooses $N_1 = N_2 = 256$. The main arithmetic NTT unit consists of a 256-point NTT (that computes the inner N_1-point NTT and outer N_2-point NTT) followed by 255 post-multipliers (that multiply with the twiddles $\omega_N^{n_2 k_1}$). We employ a standard DIF flow graph for the 256-point NTT, where we replace multiplications $\omega^0 = 1$ with simple pipeline balancing registers. Through this optimization, the $N = 256$-point sub-NTTs are implemented with only 769 modular multipliers, instead of $N/2 \log(N) = 1024$.

As discussed in Section 3.2, additional pre- and post-multiplication steps are required to construct negacyclic forward and inverse NTTs from regular NTTs. Because a radix-256 butterfly already includes an array of 255 post-multipliers, it suffices to add 255 pre-multipliers to support negacyclic NTTs. Figure 3 illustrates how a radix-4 unit is composed to compute the full NTT flow graph in two passes that each take 4 chunks. In between the passes is an implicit memory transposition that we enable with a *conflict-free CTB design*.

Our NTT PE instantiates four parallel 3-stage NTT units. Each unit is deeply pipelined with 40 pipeline stages in order to run at 2 GHz. Together, these four parallel pipes consume 1024 32-bit residue polynomial coefficients at that 2 GHz rate – sufficient to consume all available data bandwidth from the CTB.

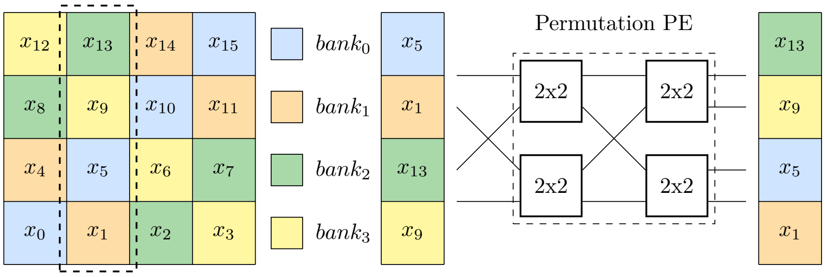

Figure 4: Example conflict-free CTB layout for a 16-point radix-4 NTT. Data is striped using the equation $bank = row \oplus col$, which ensures that both entire columns or entire rows can be read out without bank conflicts. The on-the-fly Permutation PE maps values from *bank order* into *natural order*, as illustrated for access to the second column.

7.1.1 Conflict-Free Schedule

A well-known performance inhibitor for NTTs is that successive NTT passes access coefficients at different memory strides, introducing access conflicts in memory. Prior NTT accelerators present custom access patterns and reordering techniques that only work for small-radix NTT architectures [RVM$^+$14, RLPD20] or require expensive in-memory transpositions [SFK$^+$21]. BASALISC avoids reinventing the wheel, instead building upon years of DSP literature [Coh76, Joh92, Ma99]. The most high-performance FFT accelerators present *conflict-free schedules* [TL11, RV08, RMD$^+$15] to tackle this exact issue.

Conceptually, a $N = N_1N_2 = 256^2$-point radix-256 NTT can be represented as a two-dimensional NTT, where the data is laid-out with $N_1 = 256$ rows and $N_2 = 256$ columns. In this format, the inner N_1-point NTT requires coefficients in column-major order, whereas the outer N_2-point NTT requires data in row-major order. The crux of building conflict-free NTT schedules is to structure the data so that it can be read out in either order without bank conflicts. This requires a minimum of 256 independently addressable banks, each containing 2^{16} bank addresses (for a total CTB size of 2^{24} values).

We employ a conflict-free layout based on XOR-permutations [RMD$^+$15], as illustrated in Figure 4. In this layout, data with logical address $\{row, col\}$ is stored at $bank = row \oplus col$. This layout ensures that each unique index for every element in every row and column corresponds to a unique physically accessible bank of CTB SRAM.

When reading rows or columns from the CTB, values come out of memory in *bank order*, one value for each bank from bank 0 to 255. However, operations like NTT require values in *natural order*: when accessing a row, we need values sorted by column from 0 to 255, and when accessing a column, we need values sorted by row from 0 to 255. Thus, when accessing row r, we must map bank i to index $i \oplus r$. Likewise, when accessing column c, we must map bank i to index $i \oplus c$.

We build a custom "on-the-fly" Permutation PE to compute these XOR-based permutations as data moves to or from the other PEs. Furthermore, we observe a remarkable optimization opportunity for this unit. By implementing a slightly more general permutation PE that supports permutations of the form $i \mapsto (i \cdot a + b) \oplus c$, we can not only use the Permutation PE to implement conflict-free XOR permutations, but also any BGV ring automorphism *without additional hardware*. See Section 7.2 for more details.

7.1.2 Twiddle Factor Factory

Similarly to polynomial residue coefficients, twiddle factors are 32-bit integers. There are N twiddle factors for each residue for both forward and inverse NTT, and a maximum of 56 residues at max-capacity key switching, together requiring ∼29.4 MB of twiddle factor material in a naive implementation. Moreover, our four NTT units have 5116 multipliers total that must be fed each cycle with twiddles, requiring massively parallel access into this storage memory. BASALISC prevents this storage requirement in two ways. First, we contribute new insights and a twiddle decomposition method that reduces the required parallel number of distinct twiddle accesses. Second, we develop a custom *twiddle factor factory* that drastically reduces the number of twiddles stored. In the remainder, we analyze only the forward NTT, but identical optimizations apply to the inverse NTT.

For a forward negacyclic NTT, each input x_i is pre-multiplied by the twiddle $\phi^i = \omega_{2N}^i$. Using techniques from the DSP literature [Gar16], we decompose the additional negacyclic twiddles to extract a regular pattern, and to distribute them evenly between the two NTT passes in the flow graph. This is illustrated in Figure 3 by the extra twiddles in blue. The benefit of this technique is twofold. Firstly, it can be seen that the pre-multiplications become identical for each chunk in both passes. This allows the four NTT units to share the same pre-multiply twiddles, and drastically reduces the total number of pre-multiply twiddles from $N = 256^2$ to $2 \cdot \sqrt{N}$, easily fitting in a small SRAM. Second, the internal butterfly twiddles (powers of ω_{256}) are now a strict subset of the pre-multiply twiddle in the first pass (powers of ω_{512}). Both can therefore be routed from the same small SRAM.

The remaining twiddle factor complexity sits in the post-multiply twiddles. For each chunk k, there are 255 twiddles $\omega_{256^2}^{ik}$. An SRAM storing vectors of 255 twiddles with depth 255 for each residue is still much too large. We propose a technique to reduce the *width* of this SRAM. It can be coupled with techniques that reduce the *depth* of this SRAM, such as On-the-fly-Twiddling (OT) [KJPA20]. To reduce the width, we propose a *power generator circuit* that trades SRAM storage for multipliers. The main idea is as follows. By using the identity $\omega_{256^2}^{ik} = \omega_{256^2/k}^{i}$, it can be observed that the required twiddles for chunk k are always the 255 consecutive powers of a *seed* value $\omega = \omega_{256^2/k}$. Using only ω, we can compute its successive powers in a number of multiply layers. The first layer computes ω^2 from ω, with a single multiplier. The second layer takes ω^2 and ω to compute ω^4 and ω^3, and so forth. Every multiplier in the circuit produces a unique value that is used as an output, so the number of multipliers to generate 255 powers from ω is simply 254. Using this technique, instead of storing vectors of 255 twiddles with depth 255 for each residue, it suffices to store the single seeds with depth 255.

7.2 Permutation PE

A pair of Permutation PEs forms the interface between the CTB and the other PEs. Our original contribution is a slightly more generalized Permutation PE that can support both conflict-free schedules required by NTT operations, as well as BGV automorphisms *with the same hardware.* In order to do so, the Permutation PE is generalized to compute permutations of the form $i \mapsto (i \cdot a + b) \oplus c$. Each permutation unit reorders an array of input coefficients to produce a permuted output array of the same length.

The *Read Permutation PE* unscrambles data in conflict-free CTB bank ordering in order to pass it to the other PEs expecting natural ordering. It is a specialized instance of the more general Permutation PE that only implements permutations $i \mapsto i \oplus c$, requiring values $a = 1$ and $b = 0$. The *Write Permutation PE* passes data in the opposite direction. It implements the general permutation $i \mapsto (i \cdot a + b) \oplus c$ in order to re-scramble the data into its conflict-free layout, or to compute ring automorphisms. In the latter case, the output of the Read Permutation PE is fed directly into the input of the Write Permutation PE to achieve the complete operation of the automorphism.

Each Permutation PE itself is split into two portions. Firstly, the data-permutation portion of the logic is implemented using 2×2 switch nodes placed using an Omega-Network topology. Secondly, a configuration portion takes constants a, b, and c in order to generate the routing pattern for the switches in the network. The configuration portion of the logic attaches the routing pattern to the data and the combined payload word is sent through the network. The switch nodes forward the data according to the least significant bit of the pattern part of the payload data, which is also removed before forwarding. Thus the message is reduced by one bit at each stage of the network, and at the end, the payload only contains the data portion.

7.3 Multiply-Accumulate PE

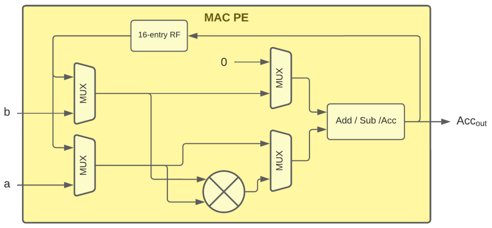

Figure 5: Simplified Diagram of MAC PE architecture.

We realize modular addition and multiplication for FHE in the Multiply-Accumulate (MAC) PE, shown in Figure 5. This pipelined unit can start 2048 32-bit modular addition or modular multiply operations each cycle, if data is available. Because the MAC PE is built with asynchronous logic, it free-runs at 1.6 GHz when not accessing the 1 GHz CTB. Therefore, operations that read and/or write to the CTB are limited by the 1 GHz CTB bottleneck, while other operations that operate on local data (accumulator register or register file) can accelerate to 1.6 GHz, without using additional logic. The asynchronous logic provides significant area and latency savings, compared to wide Clock Domain Crossings (CDCs) that would be necessary to achieve this 60% frequency increase using a clocked approach instead. At left in the figure, the 2048 a inputs - each 32-bit in size - come from the CTB. The b inputs are replicated copies of a 32-bit constant from the instruction stream. The MAC has a 16-entry Register File (RF), shown at top in the figure. In addition, there is a single accumulator register at the output of the adder/subtractor/accumulator unit, shown at right in the figure.

Using the multiplexers shown in the figure, this arrangement can accomplish a variety of functions. Residue chunk multiplication by or addition of a constant to each coefficient can be accomplished at full rate: 2048 32-bit operations per cycle. Multiplication or addition of chunks when both are sourced directly from the CTB can be accomplished at half-rate, using a register to buffer one operand from the CTB, and directly feeding the second operand into the operation from the CTB in a second read cycle. Acceleration of tight kernels that repeatedly process the same chunks can be achieved by storing up to 16 different chunks in the RF and operating on them at full rate. Finally, there is a multiply-add capability that allows double-rate processing: a multiply and accumulate in every cycle. The above possibilities are impacted by the needed write bandwidth to the CTB. Write operations might occur as often as for every chunk result, or much less often when the local RF or the accumulator store results during tight kernel operations.

A particularly important example of kernel acceleration in the MAC is key switching, which typically makes up the large majority of the workload of an FHE program. We implement the so-called *hybrid key switching* algorithm [KPZ21], which uses the “fast base extension” subroutine from Equation 3 as an inner loop. For the parameters of Table 1, fast base extension first pre-computes a table of 12 residue polynomials, and then computes 40 different weighted sums of those values, with constant weights. Use of the local registers in the MAC PE and the compound multiply-accumulate function realizes a 44× improvement compared to a naive design. In addition, this approach reduces the CTB usage from nearly 100% to only 10.6%, saving 90% of the CTB for use by the other PEs.

7.4 Modular Multiplier Arithmetic Optimization

Both the MAC and NTT butterfly units use Montgomery modular arithmetic, optimized for NTT-friendly primes (see Mert et al. [MÖS19]). Specifically, instead of supporting the full 32-bit prime value, the multiplier is optimized to only support a subset where the lower 17 bits of the prime are fixed (bits 16:1 are tied to 0 and bit 0 is tied to 1). This optimization saves 46% in area and 40% in power consumption compared to a generic multiplier that can support all moduli, and is enabled by our NTT-friendly bootstrapping approach from Section 4. The results are summarized in Table 4.

Table 4: Area and power comparison of NTT single-butterfly unit with original and optimized Montgomery multipliers at 1 GHz.

Mult. Design	Area	TDP @0.72V, 125C
Unoptimized	3768 μm^2	7.2 μW
Optimized	2052 μm^2	4.3 μW

8 BASALISC Compilation and Simulation Tools

A major component of BASALISC is co-design of software and hardware with the intent of realizing optimal performance. Our software tools include a domain-specific compiler Artemidorus and a simulator Simba, working on multiple levels of abstraction.

8.1 Artemidorus

Our toolchain begins with a high-level Domain-Specific Language (DSL) that allows programmers to create FHE applications to execute on BASALISC, and which features data types including fixed-point numbers, vectors, and matrices. The program passes through several stages in our compiler Artemidorus.

Programs written in the Artemidorus DSL are first translated into high-level FHE circuits. The toolchain is able to type-check these circuits and combinations of circuits, to make sure the overall computation is well-formed. From this circuit representation, Artemidorus infers statistics such as circuit depth, number of bit operations, and so on. Next, these circuit statistics are used to expand vector and matrix operations into BGV primitives; key and modulus switching operations are inferred by the tool; each operation is tagged with the length of its modulus chain (i.e., the number of prime factors in q), and then expanded into primitive operations on individual residue polynomials for each factor in the modulus. Finally, using a cycle-accurate model of the BASALISC microarchitecture, Artemidorus allocates memory regions and registers and schedules instructions. Instruction traces are produced at our three levels of the ISA, which pass to Simba for performance or correctness simulation. Especially *Simba-micro*, the micro-level performance simulator, is essential to evaluate BASALISC at this point in the design stage.

8.2 Simba-micro

Our micro-level performance simulator employs a step-based operational semantics to model the execution of the BASALISC coprocessor. There are five basic operational components: the CTB, and the four PEs (MAC, Read Permutation, NTT, and Write Permutation). The simulator models a micro-instruction's life cycle from instruction dispatch, to data transfer from CTB to the appropriate functional unit, to proceeding down the pipeline, to the "writeback" phase.

In order to account for the different clock rates of the different components (see Table 5), we use a global "micro-clock" which operates at 6 GHz as the time increment for the model's step function. We made the simplifying assumption that the MAC operates at 1.5 GHz. In this way, we were able to model each PE's progress by causing the CTB to be accessible every 6 micro-cycles, the MAC every 4 micro-cycles, and the permutation/NTT units every 3 micro-cycles. This behavior is modeled by supplying each component with a wait counter that is reset to these values every time it is accessed; the component is only

Table 5: Performance characteristics of BASALISC hardware elements.

Component	f_{max}	Area	TDP @0.72V	Throughput
CTB†	1.0 GHz	77.9 mm^2	9 W	2 × 32 Tb/s
MAC PE	1.6 GHz	7.17 mm^2	18.6 W	102 Tb/s
NTT PE	2.0 GHz	16 mm^2	24.6 W	32 Tb/s
Permuation PE	2.0 GHz	0.16 mm^2	∼0 W	32 Tb/s
PCIe	500 MHz	12 mm^2	5 W	26 GB/s
DDR	800 MHz	18.2 mm^2		51 GB/s
Overall	> 1.0 GHz	150 mm^2	57.5 - 115 W	N/A

†Operation of PEs above the CTB's frequency is advantageous when each can run independently.

accessible if the counter is 0. If a component is accessible but has no work to do in the given micro-cycle, it simply waits.

Each individual PE is modeled as a pipeline with a certain number of stages and "stage capacity" (number of coefficients that fit in each pipeline stage). The MAC's stage capacity is 2048 coefficients, while the other three have a capacity of 1024. Every time the given PE is enabled (its wait counter is 0), the pipeline advances. When there is a write at the end of the pipeline, it stays at the end of the pipeline until the CTB is available for writing. Once a pipeline is full, instruction dispatch is no longer possible to that pipeline, and the control mechanism allows the pending writes to occur.

After execution, the following data is reported by Simba-micro: number of CTB cycles (i.e., 6 micro-cycles) of execution, overall CTB utilization (percentage of time spent reading/writing/stalling), and utilization of each PE (how "full" the pipelines are).

9 Evaluation of BASALISC

BASALISC is a closed-source architecture with an implementation-in-progress but not delivered to silicon yet. For that reason, we evaluate BASALISC in diverse ways at this point in the design cycle.

9.1 Physical Realizability

One way to evaluate the design of BASALISC and the BASALISC architecture is by a physical design implementation in a practical semiconductor process, with a reasonable target die size and operational frequency target. One resulting evaluation criterion that can be objectively measured using this approach is timing closure – the verification that, with placement and routing of key blocks complete, and using industry best practice estimation of inter-block wire delays based on a mature floorplan, the design achieves a target operating frequency that yields useful levels of performance. In the case of BASALISC, we completed placement and routing of the novel circuitry - our PEs - and the CTB RAM block. Our operational frequency target was a minimum of 1.0 GHz at the standard "slow-slow" (SS) process corner and a supply voltage of 0.72V in the 12nm low-power Global Foundries process. We achieved timing closure for the diverse functional units at the frequencies given in Table 5.

Our floorplan shown in Figure 6 uses actual IP block sizes for DDR4 DRAM, PCIe, our RAM array, and placed and routed PEs. I/O pads are part of the DDR4/PCIe macros, including bumps for power and ground. Interface clock trees run along provisioned routing channels; other big clock trees are avoided through asynchronous design. Everything is drawn assuming 75% density, with a 200μm peripheral gap accounting for process DRC rules, including crackstop, corners, and ESD protection. Our target die size is constrained to 150mm^2 with an aspect ratio of 2 : 1 or less [Ron20], which we achieve with a 14.4mm × 10.4mm layout that satisfies both of those constraints.

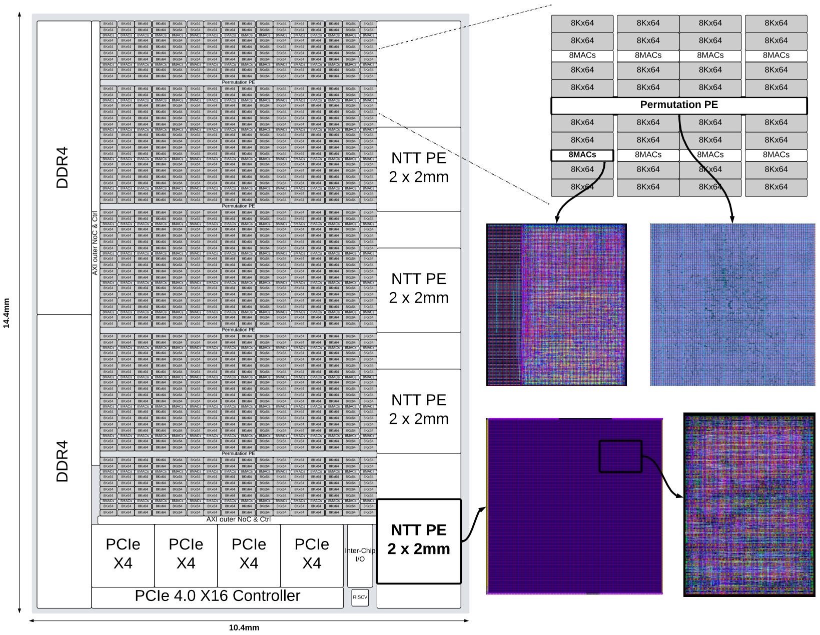

Figure 6: Floorplan of BASALISC with all cells placed and intra-block routing complete. Note that the MAC PE and Permutation PE are interleaved within the CTB.

9.2 Logic Emulation and Formal Verification

A commonplace verification step prior to ASIC manufacturing provides yet another evaluation criterion: successful *hardware emulation* of critical logic in the design. In the case of BASALISC, that critical logic is the set of processing elements (MAC, permutation, and NNT PEs). We successfully emulated each PE in full, using test vectors extracted from our Verilog testbenches and our formal models of each PE. Each PE passed its emulation test vector suite.

In addition to hardware emulation, BASALISC employs formal methods with two primary goals: first, that the design be proven mathematically correct, and second, that the design be proven consistent at every intermediate representation by demonstrating proof of equivalence. For both the mathematical and consistency proofs, BASALISC employs the Cryptol language [LM03] and related tools. In order to satisfy mathematical correctness, top-level FHE algorithms are expressed as a mathematical model in Cryptol. Subsequently, using Cryptol's proof capabilities, the mathematical model is proven to sustain a set of separately-developed correctness definitions. Proof of equivalence is provided through a two-step approach. First, formal equivalence is proven between the high-level mathematical Cryptol model and a low-level logic-oriented Cryptol description using SAW [CFH+13]. Next, the low-level Cryptol is converted to Verilog that we prove equivalent to the optimized implementation-Verilog using the commercial Synopsys Formality tool.

9.3 Benchmark Performance Simulation

We evaluate BASALISC's performance on a set of benchmarks. Table 6 summarizes the results and compares to an HElib software reference, which was executed on an Intel Xeon E5-2630 v2 CPU at 2.6 GHz with a single thread. All results are generated using the parameter set from Table 1 (only the plaintext modulus for database lookup is chosen differently for comparison reasons). This results in ciphertext size 21 MB and key switching keys of size 56 MB.[1]

The first part of Table 6 compares BASALISC performance to HElib for a set of *micro-benchmarks*: a ciphertext NTT and the basic and auxiliary homomorphic operations. Each operand is a freshly encrypted ciphertext. We achieve major speedups for all homomorphic operations. In particular, we accelerate key switching - the most time-intensive operation - by a factor of $2.0 \cdot 10^3\times$.

Table 6: Performance comparison of HElib and BASALISC.

Operation	HElib	BASALISC	Speedup
NTT	27 ms	11 μs	$2.5 \cdot 10^3\times$
Add/Sub	4 ms	8 μs	$5.0 \cdot 10^2\times$
Plaintext mul	44 ms	5 μs	$8.8 \cdot 10^3\times$
Mul (no key switch)	58 ms	20 μs	$2.9 \cdot 10^3\times$
Permutation (no key switch)	12 ms	11 μs	$1.1 \cdot 10^3\times$
Key switching	580 ms	292 μs	$2.0 \cdot 10^3\times$
Database lookup†	2,325 s	267 ms	$8.7 \cdot 10^3\times$
Thin bootstrapping	160 s	40 ms	$4.0 \cdot 10^3\times$
Logistic regression	217,000 s	40,500 ms	$5.4 \cdot 10^3\times$

†Simulation done with $t = 241$ to make the result comparable with F1 [SFK+21].

We also simulated execution time for three realistic *macro-benchmarks*: a database lookup, a bootstrapping operation and a single iteration of encrypted logistic regression training. The last two benchmarks require bootstrapping. This is done with HElib's thin bootstrapping procedure [HS21], but adapted to our NTT-friendly approach.

- **Database lookup.** This application comes from the HElib repository [HEl]. An encrypted key is sent from a client to a server, then the server compares it homomorphically against a database of encrypted key-value pairs. Finally, the corresponding value is returned to the client in encrypted format. We achieve a speedup of 8,700 times over HElib for the server computation.

- **Thin bootstrapping.** This benchmark is evaluated with bootstrapping parameter $e = 4$ (see Section 4). Bootstrapping requires in total 26 key switching keys, which are loaded from DRAM into the CTB during the procedure. Simba-micro reports only 40ms of simulated execution time, which is a speedup of 4,000 times over HElib. The required noise budget before bootstrapping is 47 bits to account for a linear transformation and correctness of decryption. The remaining noise budget after bootstrapping is 790 bits, which gives 18 multiplicative levels between successive bootstrapping operations.

- **Logistic regression.** We estimate execution time on a single iteration of secure logistic regression (LR) training, using the 1-bit gradient descent algorithm of Chen et al. [CGBH+18]. This application homomorphically trains a machine learning model on a 1,024-sample, 10-feature infant mortality data set from the US Centers for Disease Control. As a BASALISC instruction trace, logistic regression is composed of 900K high-level, 800M mid-level, and 27B micro-level instructions, which includes in total 513 bootstrapping operations. Simba-micro reports a simulated execution time of 40.5s. Because logistic regression is not in HElib, software execution time is estimated by counting individual operations. Since this benchmark is bootstrapping-dominated, the obtained speedup is similar to thin bootstrapping.

[1] Each key switching key is a matrix in $\mathcal{R}_{QP}^{d\times 2}$, where the benchmarks use $d = 4$. This results in a key size of $2d \cdot N \cdot \log_2(QP) \approx 112$ MB. However, the second column is uniformly distributed, so instead of storing it, we keep a seed and generate it on the fly. This technique is standard and also used in HElib [HS20].

Table 7: Comparison of BASALISC with different BGV/CKKS accelerators.

	BASALISC	F1	BTS	CraterLake
Scheme	**BGV**	BGV/CKKS	CKKS	CKKS
Area	**150mm^2**	150mm^2	374mm^2	472mm^2
Technology	**12nm**	12/14nm	7nm	12/14nm
Power	**115W**	180W	163W	320W
Bootstrapping speedup	**4,000**×	1,830×/1,195×	2,237×	4,400×
LR speedup[†]	**5,400**×	—/7,200×	1,306×	2,978×

[†]Using [CGBH+18] for BGV LR and [HHCP19] for CKKS LR.

BASALISC is optimized for the default ring dimension $N = 2^{16}$, which enables bootstrapping with a 128-bit security target. Smaller parameter sets are insufficient to support bootstrapping at this security level, but they can be useful in leveled applications of low multiplicative depth. BASALISC's speedup tends to be somewhat smaller for smaller ring dimension, but it is still significant. For example, database lookup is $2,600\times$ faster for $N = 2^{14}$, and thin bootstrapping with $N = 2^{15}$ achieves $2,500\times$ speedup.

9.4 Related Work

We perform a comparison to prior and concurrent FHE accelerators. Many early works do not report bootstrapping benchmarks or simply do not support it [PNPM15, MAK+22, CMV+15, DÖS14, SRTJ+19, SRJV+18, SRJV+15, RLPD20, TRV20, ABK20, SYYT20]. These architectures support only unrealistically small parameter sets, often allowing them to fully compute on-chip. Furthermore, not all accelerators implement full FHE computations, but rather individual operations such as the NTT. As one outcome, these other approaches require frequent interaction with a host processor to sequence operations and combine results. In this category of accelerators, HEAX [RLPD20] achieves the most siginificant acceleration numbers, in the order of 200× for high-level operations such as key switching, compared to SEAL [SEA22].

Other accelerators that support bootstrapping implement the homomorphic scheme CKKS. Although CKKS and BGV are very similar, there are important low-level differences, and an accelerator for one scheme may not necessarily support the other. BASALISC's comparison to the related BGV/CKKS accelerators is summarized in Table 7.

BTS [KKK+21] is a CKKS accelerator with a large 374mm^2 area budget (2.5× ours). BTS uses a grid-based microarchitecture that lays out 2,048 PEs as 32 by 64. Conceptually, this architecture is much more complex than the simple vector architecture of BASALISC. Each PE unit has a local SRAM memory, an NTT unit, a "base extension" unit, adders, and multipliers. This incurs a lot of communication between the PEs, so to simplify the data movement management, they treat the entire output of each PE as a "package of coefficients". This restricts the automorphisms that BTS can evaluate to the shape $c(X) \mapsto c(X^{5^i})$. While sufficient for CKKS, BTS cannot compute all automorphisms for BGV bootstrapping. BTS uses On-the-fly Twiddling (OT) [KJPA20] to store twiddle factors. As we mentioned in Section 7.1.2, this technique can be further combined with our more efficient twiddle factor factory to drastically reduce the requirements of twiddle factor storage even more. Even at the larger area budget, BTS reports similar speedups to BASALISC. Respectively, in a first benchmark involving ciphertext multiplication and bootstrapping, and a second logistic regression training benchmark, BTS outperforms the Lattigo software library by 2,237× and 1,306×.

The architecture that is closest to ours and also supports BGV is F1 [SFK+21]. F1 is an ASIC architecture targeting the same die size (150mm^2), technology node (12nm GF) and clock frequency (1 GHz). F1 reports a bootstrapping time of 2.4ms, compared to 40ms for our macro-benchmark. However, these numbers are not directly comparable: F1 provides lower security (4× smaller ring dimension N) and bootstraps a plaintext space of only 1 bit with no packing, whereas our benchmark has plaintext modulus 127^3 with packing capability. The authors also report execution time for database lookup, but again with smaller ring dimension. F1's speedup for this application - around $7,000\times$ - is comparable to ours. Although F1 is programmable and could support packed bootstrapping, it does not have enough multiplicative levels available to run it at a realistic security level. Moreover, F1 scales poorly to larger parameter sets: it is optimized for simple BV key switching, which is less efficient than hybrid key switching for high-depth computations [KPZ21]. Our MAC PE with local RF is essential to accelerate hybrid key switching.

A novel aspect of our accelerator is the conflict-free CTB and NTT, with the corresponding Permutation PE that reuses the same hardware for NTT and automorphism. Compared to F1's FFT algorithm, we avoid an expensive matrix transpose unit by computing the same transpose directly within the CTB. F1's matrix transpose unit must buffer entire ciphertext polynomials within the NTT PE, which is prohibitive for our parameter set. Whereas BASALISC includes a highly-optimized twiddle factor factory, F1 does not describe how to implement its large twiddle factor SRAM.

A recently proposed follow-up work to F1 is CraterLake [SFK+22]. CraterLake makes two improvements over F1 that we considered from the onset, but which BASALISC supports more efficiently. Firstly, CraterLake adds support for hybrid key switching through a complex "vector chaining" technique. BASALISC supports the same algorithm efficiently within its simple MAC unit with an integrated 16-entry register file. Secondly, CraterLake observes that F1's SRAM NTT matrix transpose does not scale to larger parameters sets. It therefore decomposes this into a smaller intra-lane-group SRAM transpose, and an extra inter-lane-group fixed permutation network, both distinct from the central register file. BASALISC leverages the existing CTB memory infrastructure to compute transposes without additional units or memory, using its conflict-free schedule based on XOR-permutations.

CraterLake is a substantially larger chip (472mm^2, 320W) than BASALISC, and only evaluates CKKS. It accelerates CKKS packed bootstrapping by 4,400 times, which is similar to our speedup for BGV packed bootstrapping. At the same time, BASALISC appears closer to tape-out: CraterLake does not formally verify its PEs, does not present a layout or floorplan, and includes difficult-to-manufacture IP cores such as HBM2E.

10 Conclusion

FHE enables new privacy-preserving applications, but its adoption is limited because of high computational costs. BASALISC accelerates FHE computations by more than three orders of magnitude over CPUs, and thereby takes a step toward practical feasibility.

In contrast to prior works, BASALISC supports all BGV operations, including fully-packed bootstrapping, in a single ASIC architecture. Our design includes a complete memory hierarchy, and an ISA that supports different levels of abstraction. We propose several hardware improvements in the NTT architecture, and show that its permutation unit can be generalized to compute BGV automorphisms without additional area. BASALISC saves area and power consumption by restricting its multipliers to NTT-friendly primes. This optimization still allows BGV bootstrapping, so it does not compromise generality.

We evaluate the design of BASALISC for correctness and performance. Our functional units are emulated and formally verified to meet their specification. We also simulate performance on three FHE macro-benchmarks, showing over 5,000 times speedup compared to classical software implementations. BASALISC has been selected as a candidate for future fabrication of an IC that can be applied in real-world applications.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-21-C-0034. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This work was additionally supported in part by CyberSecurity Research Flanders with reference number VR20192203 and the Research Council KU Leuven (C16/15/058). Michiel Van Beirendonck is funded by Research Foundation – Flanders (FWO) as Strategic Basic (SB) PhD fellow (project number 1SD5621N). Robin Geelen is funded in part by Research Foundation – Flanders (FWO) under a PhD Fellowship fundamental research (project number 1162123N).

References

[ABK20] Rashmi S. Agrawal, Lake Bu, and Michel A. Kinsy. Fast arithmetic hardware library for rlwe-based homomorphic encryption. In *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2020, Fayetteville, AR, USA, May 3-6, 2020*, page 206. IEEE, 2020.

[ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[ASP13] Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In *Annual Cryptology Conference*, pages 1–20. Springer, 2013.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BIP+22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. Final: Faster fhe instantiated with ntru and lwe. Cryptology ePrint Archive, Report 2022/074, 2022. https://ia.cr/2022/074.

[CFH+13] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb. Saw: The software analysis workbench. In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, page 15–18, New York, NY, USA, 2013. Association for Computing Machinery.

[CGBH+18] Hao Chen, Ran Gilad-Bachrach, Kyoohyung Han, Zhicong Huang, Amir Jalali, Kim Laine, and Kristin Lauter. Logistic regression over encrypted data from fully homomorphic encryption. *BMC medical genomics*, 11(4):3–12, 2018.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CH18] Hao Chen and Kyoohyung Han. Homomorphic lower digits removal and improved fhe bootstrapping. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 315–337. Springer, 2018.

[CJL$^+$20] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, volume 15, 2020.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[CMV$^+$15] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Pao, and Ingrid Verbauwhede. High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(1):157–166, January 2015.

[Coh76] D. Cohen. Simplified control of fft hardware. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24(6):577–579, 1976.

[DÖS14] Yarkın Doröz, Erdinç Öztürk, and Berk Sunar. Accelerating fully homomorphic encryption in hardware. *IEEE Transactions on Computers*, 64(6):1509–1521, 2014.

[DSL$^+$18] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[Gar16] Mario Garrido. A new representation of fft algorithms using triangular matrices. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(10):1737–1745, 2016.

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

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P Smart. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 465–482. Springer, 2012.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012.

[GV22] Robin Geelen and Frederik Vercauteren. Bootstrapping for bgv and bfv revisited. *Cryptology ePrint Archive*, 2022.

[HEl] HElib country lookup example. https://github.com/homenc/HElib/tree/master/examples/BGV_country_db_lookup. Accessed: 2022-12-17.

[HHCP19] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 9466–9471. AAAI Press, 2019.

[HS14] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.

[HS20] Shai Halevi and Victor Shoup. Design and implementation of helib: a homomorphic encryption library. *Cryptology ePrint Archive*, 2020.

[HS21] Shai Halevi and Victor Shoup. Bootstrapping for helib. *Journal of Cryptology*, 34(1):1–44, 2021.

[Joh92] L.G. Johnson. Conflict free memory addressing for dedicated fft hardware. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(5):312–316, 1992.

[KJPA20] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*, pages 264–275. IEEE, 2020.

[KKK$^+$21] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, Minsoo Rhu, John Kim, and Jung Ho Ahn. BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption. *arXiv:2112.15479 [cs]*, December 2021.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 608–639, Cham, 2021. Springer International Publishing.

[Lat22] Lattigo v3. Online: https://github.com/tuneinsight/lattigo, April 2022. EPFL-LDS, Tune Insight SA.

[LFS87] Kun-Shan Lin, G.A. Frantz, and R. Simar. The tms320 family of digital signal processors. *Proceedings of the IEEE*, 75(9):1143–1159, 1987.

[LM03] Jeffrey R Lewis and Brad Martin. Cryptol: High assurance, retargetable crypto development and validation. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, volume 2, pages 820–825. IEEE, 2003.

[Ma99] Yutai Ma. An effective memory addressing scheme for FFT processors. *IEEE Trans. Signal Process.*, 47(3):907–911, 1999.

[MAK+22] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. Medha: Microcoded hardware accelerator for computing on encrypted data. Cryptology ePrint Archive, Report 2022/480, 2022. https://ia.cr/2022/480.

[MÖS19] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savas. Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. In *22nd Euromicro Conference on Digital System Design, DSD 2019, Kallithea, Greece, August 28-30, 2019*, pages 253–260. IEEE, 2019.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating Homomorphic Evaluation on Reconfigurable Hardware. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, Lecture Notes in Computer Science, pages 143–163, Berlin, Heidelberg, 2015. Springer.

[PRR17] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. Palisade lattice cryptography library user manual. 2017.

[RAD+78] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[RLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1295–1309. ACM, 2020.

[RMA+21] Sujoy Sinha Roy, Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, and Donghoon Yoo. Accelerator for computing on encrypted data. *IACR Cryptol. ePrint Arch.*, page 1555, 2021.

[RMD+15] Stephen Richardson, Dejan Marković, Andrew Danowitz, John Brunhaver, and Mark Horowitz. Building conflict-free fft schedules. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(4):1146–1155, 2015.

[Ron20] Tom Rondeau. Data protection in virtual environments (DPRIVE), 2020.

[RV08] Dionysios I. Reisis and Nikolaos Vlassopoulos. Conflict-free parallel memory accessing techniques for FFT architectures. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 55-I(11):3438–3447, 2008.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014.

[SEA22] Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL, March 2022. Microsoft Research, Redmond, WA.

[SFK+21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 238–252, New York, NY, USA, 2021. Association for Computing Machinery.

[SFK+22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 173–187. ACM, 2022.

[SRJV+15] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, and Ingrid Verbauwhede. Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, Lecture Notes in Computer Science, pages 164–184, Berlin, Heidelberg, 2015. Springer.

[SRJV+18] Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation. *IEEE Transactions on Computers*, 67(11):1637–1650, November 2018.

[SRTJ+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398, February 2019.

[SV14] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[SYYT20] Yang Su, Bailong Yang, Chen Yang, and Luogeng Tian. Fpga-based hardware accelerator for leveled ring-lwe fully homomorphic encryption. *IEEE Access*, 8:168008–168025, 2020.

[TL11] Pei-Yun Tsai and Chung-Yi Lin. A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling. *IEEE Trans. Very Large Scale Integr. Syst.*, 19(12):2290–2302, 2011.

[TRV20] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. HEAWS: an accelerator for homomorphic encryption on the amazon AWS FPGA. *IEEE Trans. Computers*, 69(8):1185–1196, 2020.

[Zuc18] Vincent Zucca. *Towards efficient arithmetic for Ring-LWE based homomorphic encryption.* PhD thesis, Sorbonne université, 2018.