FPT: a Fixed-Point Accelerator for Torus Fully Homomorphic Encryption

Michiel Van Beirendonck, Jan-Pieter D'Anvers, Ingrid Verbauwhede
{firstname.lastname}@esat.kuleuven.be
COSIC KU Leuven
Leuven, Belgium

ABSTRACT

Fully Homomorphic Encryption is a technique that allows computation on encrypted data. It has the potential to drastically change privacy considerations in the cloud, but high computational and memory overheads are preventing its broad adoption. TFHE is a promising Torus-based FHE scheme that heavily relies on bootstrapping, the noise-removal tool that must be invoked after every encrypted gate computation.

We present FPT, a Fixed-Point FPGA accelerator for TFHE bootstrapping. FPT is the first hardware accelerator to heavily exploit the inherent noise present in FHE calculations. Instead of double or single-precision floating-point arithmetic, it implements TFHE bootstrapping entirely with approximate fixed-point arithmetic. Using an in-depth analysis of noise propagation in bootstrapping FFT computations, FPT is able to use noise-trimmed fixed-point representations that are up to 50% smaller than prior implementations using floating-point or integer FFTs.

FPT's microarchitecture is built as a streaming processor inspired by traditional streaming DSPs: it instantiates high-throughput computational stages that are directly cascaded, with simplified control logic and routing networks. We explore different throughput-balanced compositions of streaming kernels with a user-configurable streaming width in order to construct a full bootstrapping pipeline. FPT's streaming approach allows 100% utilization of arithmetic units and requires only small bootstrapping key caches, enabling an entirely compute-bound bootstrapping throughput of 1 BS / 35μs. This is in stark contrast to the established classical CPU approach to FHE bootstrapping acceleration, which tends to be heavily memory and bandwidth-constrained.

FPT is fully implemented and evaluated as a bootstrapping FPGA kernel for an Alveo U280 datacenter accelerator card. FPT achieves almost three orders of magnitude higher bootstrapping throughput than existing CPU-based implementations, and 2.5× higher throughput compared to recent ASIC emulation experiments.

1 INTRODUCTION AND MOTIVATION

Machine Learning (ML), driven by the availability of an abundance of data, has seen rapid advances in recent years [48], leading to new applications from autonomous driving [63] to medical diagnosis [38]. In many applications, ML models are developed by one party, who makes them available to users as a cloud service [3]. The deployment of such applications comes at the risk of privacy breaches, where the user data might be leaked, or IP theft, where users steal the ML model from the developing party [46].

The "silver bullet" solution to prevent the leakage of this data is to encrypt it with Fully Homomorphic Encryption (FHE) [27, 50], which is a technique that allows one to compute on encrypted data.

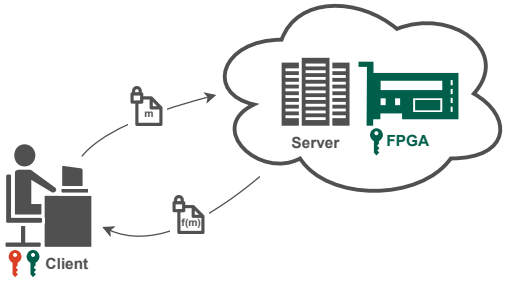


Figure 1: FHE allows outsourced computation on data that remains encrypted. The cloud receives encrypted data on which it can compute and the public key (green), but does not receive the secret decryption key (red). The cloud can run computations on the data, but only the client can finally decrypt and obtain the result. Cloud instances with FPGAs enable custom hardware accelerators and have the potential to drastically speed up FHE computations.

Fig. 1 illustrates a possible application of FHE to protect user data in an ML environment. In this scenario, a client wants to use an online-server-based ML service, without leaking any sensitive data. To this end, the client encrypts their data with FHE, before sending it to the cloud. The cloud service then computes an FHE program on the encrypted data without obtaining any information about the input and sends the (still encrypted) result back to the client. Only the client can finally decrypt and obtain the result.

The drawback of FHE is that it is at the moment still orders of magnitude slower than unencrypted calculations. The first algorithm to calculate an encrypted AND gate took up to 30 minutes to finish[28]. FHE schemes and algorithms have seen significant improvements in recent years, e.g. the recent TFHE scheme computes encrypted AND gates in only 13ms [10, 11] on a CPU. However, even with these improvements, it is not uncommon to still see slowdown factors of 10,000× compared to calculations on unencrypted data [13, 31, 39], which currently still prevents practical deployment of FHE in many applications.

To work around the speed limitations of FHE, designers have shifted their focus from general-purpose CPUs to more dedicated hardware implementations. Of these dedicated implementations, GPU-based FHE accelerators are easiest to develop, but they typically only provide modest speedups [5, 15, 35, 58]. At the other end of the spectrum, ASIC emulations in advanced technology nodes promise better FHE acceleration [25, 36, 37, 52, 53]. However, it can take years for these ASICs to be fabricated and become available

[44], and they are typically specialized for a limited range of parameter sets. Finally, FPGA-based implementations can be developed more quickly than ASIC implementations, are flexible to change parameter sets, and can be readily deployed in FPGA-equipped cloud instances while boosting large speedups. As a result, they have been a popular target for FHE acceleration [1, 18, 41, 47, 49, 51, 57].

One costly operation in FHE calculations is bootstrapping. All currently available FHE schemes have an inherent noise that is increased with each operation. After a certain number of operations, this noise needs to be reduced to allow further calculations, which is done using this so-called bootstrap procedure.

Second-generation FHE schemes BFV [20], BGV [7], and CKKS [9] have been the main focus of prior hardware accelerators. These schemes require bootstrapping only after a certain number of operations. For these schemes, bootstrapping is a complex algorithm that requires large data caches [16] and exhibits low arithmetic intensity, and essentially all prior architectures that support bootstrapping have hit the off-chip memory-bandwidth wall [37, 52].

Third-generation schemes like FHEW [19] and its successor Torus FHE (TFHE) [10, 11] have revisited the bootstrapping approach, making it cheaper but inherently linked to homomorphic calculations. In these schemes, most of the homomorphic operations require an immediate bootstrap of the ciphertext. Moreover, bootstrapping in TFHE is a versatile tool, which can additionally be "programmed" with an arbitrary function that is applied to the ciphertext, e.g. non-linear activation functions in ML neural networks [13]. This approach is called Programmable Bootstrapping (PBS) and it constitutes the main cost of TFHE homomorphic calculations. Taking up to 99% of an encrypted gate computation, PBS is a prime target for high-throughput hardware acceleration of TFHE.

In this work, we propose FPT, an FPGA-based accelerator for TFHE Programmable Bootstrapping. FPT achieves a significant speedup over the previous state-of-the-art, which is attributable to two major contributions:

- FPT's microarchitecture is built as a streaming processor, challenging the established classical CPU approach to FHE bootstrapping accelerators. Inspired by traditional streaming DSPs, FPT instantiates high-throughput computational stages that are directly cascaded, with simplified control logic and routing networks. FPT's streaming approach allows 100% utilization of arithmetic units during bootstrapping, including tool-generated high-radix and heavily optimized negacyclic FFT units with user-configurable streaming widths. Our streaming architecture is discussed in Section 3.
- FPT (**F**ixed-**P**oint **T**FHE) is the first hardware accelerator to extensively optimize the representation of intermediate variables. TFHE PBS is dominated by FFT calculations, which work on irrational (complex) numbers and need to be implemented with sufficient accuracy. Instead of using double floating-point arithmetic or large integers as in previous works, FPT implements PBS entirely with compact fixed-point arithmetic. We analyze in-depth the noise due to the compact fixed-point representation that we use inside PBS, and we match it to the noise that is natively present in FHE. Through this analysis, FPT is able to use fixed-point representations that are up to 50% smaller than prior implementations using floating-point or integer FFTs. In turn, these 50% smaller fixed-point representations enable up to 80% smaller FFT kernels. Our fixed-point analysis is discussed in Section 4.

FPT shows, for the first time, that PBS can remain entirely compute-bound with only small bootstrapping key data caches. FPT achieves a massive PBS throughput of 1 PBS / 35μs, which requires only modest off-chip memory bandwidth, and is entirely bound by the logic resources on our target Xilinx Alveo U280 FPGA. This represents almost three orders of magnitude speedup over the popular TFHE software library CONCRETE [12] on an Intel Xeon Silver 4208 CPU at 2.1 GHz, a factor 7.1× speedup over a concurrently-developed FPGA architecture [62], and a factor 2.5× speedup over recent 16nm ASIC emulation experiments [33].

2 BACKGROUND

This section gives an intuitive idea of the workings of TFHE, with a focus on the Programmable Bootstrapping step that is accelerated by FPT. We the reader to [10, 11, 34] for a more in-depth overview of TFHE.

2.1 Torus Fully Homomorphic Encryption

Torus Fully Homomorphic Encryption (TFHE) is a homomorphic encryption scheme based on the Learning With Errors (LWE) problem. It operates on elements that are defined over the real Torus $\mathsf{T} = \mathsf{R}/\mathsf{Z}$, i.e. the set $[0, 1)$ of real numbers modulo 1. In practice, Torus elements are discretized as 32-bit or 64-bit integers.

A TFHE ciphertext can be constructed by combining three elements: a secret vector s with n coefficients following a uniform binary distribution $s \xleftarrow{\$} \mathcal{U}(\mathsf{B}^n)$, a public vector $a \xleftarrow{\$} \mathcal{U}(\mathsf{T}^n)$ sampled from a uniform distribution, and a small error $e \xleftarrow{\$} \chi$ from a small distribution $\chi(\mathsf{T})$. A message $\mu \in \mathsf{T}$ can be encrypted as a tuple: $c = (a, b = a \cdot s + e + \mu) \in \mathsf{T}^{n+1}$. Using the secret s, one can decrypt the ciphertext back into (a noisy version of) the message by computing $b - a \cdot s = \mu + e$. This type of ciphertext is called a Torus LWE (TLWE) ciphertext.

TFHE additionally describes two variant ciphertexts: First, a generalized version (TGLWE), where e and μ are polynomials in $\mathsf{T}_N[X] = \mathsf{T}[X]/(X^N + 1)$, and where a and s are vectors of polynomials of the form $\mathsf{T}_N[X]^k$. The TGLWE ciphertext is then similarly formed as a tuple: $c = (a, b = a \cdot s + e + \mu) \in \mathsf{T}_N[X]^{k+1}$. The second variant is a generalized version of a GSW [29] ciphertext (TGGSW), which is essentially a matrix where each row is a TGLWE ciphertext: $c \in \mathsf{T}_N[X]^{(k+1)l \times (k+1)}$.

The reason for defining TGLWE and TGGSW ciphertexts is that they permit a homomorphic multiplication:

$$\mathrm{TGLWE}(\mu_1) \quad \mathrm{TGGSW}(\mu_2) = \mathrm{TGLWE}(\mu_1 \cdot \mu_2),$$

known as the *External Product* ( ). First, it decomposes each of the polynomials in the TGLWE ciphertext into l polynomials of β bits, an operation termed gadget decomposition. Next, the decomposed TGLWE ciphertext and TGGSW are multiplied in a $(k+1)l$−vector times $(k+1)l \times (k+1)$-matrix product where the elements of this

vector and matrix are polynomials in $\mathsf{T}_N[X]$. The output is again a TGLWE ciphertext encrypting $\mu_1 \cdot \mu_2$.

2.2 Programmable Bootstrapping

The main goal of bootstrapping is to reduce the noise in the ciphertext. One way to reduce the ciphertext noise would be to decrypt the ciphertext, after which the noise can be suppressed, but this would not be secure. Bootstrapping does in essence decrypt the ciphertext, but for security reasons this operation is performed, homomorphically, inside the encrypted domain. This means that one wants to homomorphically compute $b - a \cdot s = e + \mu$, and more specifically, as it is "programmable" bootstrapping, one wants to additionally compute a function $f(\mu)$ on the data.

To achieve this programmable bootstrapping, one first sets a "test" polynomial $F = \sum_{i=0}^{N-1} f(i) \cdot X^i \in \mathsf{T}_N[X]$ that encodes N relevant values of the function f. This polynomial is then rotated with $b - a \cdot s$ positions by calculating $F \cdot X^{-(b-a\cdot s)}$, after which the output to the function can be found on the first position of the resulting polynomial. However, all of these calculations should be done without revealing the value of s.

The high-level idea of how to achieve this is to first rewrite the above expression as follows:

$$F \cdot X^{-(b-a\cdot s)} = F \cdot X^{-b} \cdot \ddot{\prod}_{i=1} X^{a_i \cdot s_i}. \tag{1}$$

This expression can be calculated iteratively. Starting with the polynomial $ACC = F \cdot X^{-b}$, one iteratively calculates:

$$ACC \leftarrow ACC \cdot X^{a_i \cdot s_i}, \tag{2}$$

which can be further rewritten, using the fact that s_i is either zero or one, to:

$$ACC \leftarrow (ACC \cdot X^{a_i} - ACC) \cdot s_i + ACC. \tag{3}$$

However, as we can not reveal s_i, we encode the s_i value in a TGGSW ciphertext BK_i, and the ACC value in a TGLWE ciphertext, after which the expression becomes:

$$ACC \leftarrow (ACC \cdot X^{a_i} - ACC) \quad \square \quad BK_i + ACC, \tag{4}$$

using the homomorphic multiplication operation $\square$. Eq. (4) homomorphically multiplexes on the secret value s_i, and is known as the Controlled MUX (CMUX).

Collectively, the different TGGSW ciphertexts $BK_1, \ldots, BK_n$, each encrypting one secret coefficient $s_1, \cdots, s_n$, are known as the bootstrapping key. The result of the operations described above is a TGLWE accumulator ACC which is "blindly" rotated with a secret amount of $b - a \cdot s$ positions, from which the output TLWE ciphertext can be straightforwardly extracted. The computations during PBS are given in Algorithm 1.

FPT implements two parameter sets of TFHE, given in Table 1. Parameter Set I is a parameter set used by the CONCRETE Boolean library with 128-bit security [12]. Parameter Set II is a 110-bit security parameter set that has previously been employed for benchmarking purposes, allowing a direct comparison of FPT with prior work [11, 64].

Algorithm 1: TFHE's Programmable Bootstrapping

```
/* TLWE Ciphertext */
input  : c_in = (a_1, ..., a_n, b) ∈ T^{n+1}
/* TGGSW Bootstrapping Key */
input  : BK = (BK_1, ..., BK_n) ∈ T_N[X]^{n×(k+1)l×(k+1)}
/* TGLWE Test Polynomial LUT */
input  : F ∈ T_N[X]^{(k+1)}
/* TLWE Ciphertext */
output: c_out ∈ T^{kN+1}
/* Test Polynomial LUT */
ACC ← F · X^{-b}
/* Blind Rotation */
for i ← 1 to n do
    /* CMUX */
    ACC ← (ACC · X^{⌊2Na_i/q⌉} − ACC)  □  BK_i + ACC
end
return c_out = SampleExtract(ACC)
```

Parameter Set		(I)	(II)
TLWE dimension	n	586	500
TGLWE dimension	k	2	1
Polynomial size	N	512	1024
Decomp. Base Log	β	8	10
Decomp. Level	l	2	2

Table 1: Parameter Sets: (I) is a parameter set used by the CONCRETE Boolean library [12] with 128-bit security. (II) is a 110-bit security parameter set popular for benchmarking purposes[11, 64].

2.3 FFT polynomial multiplications

As can be seen in Algorithm 1, the TFHE programmable bootstrapping mainly consists of iterative calculation of the external product $\square$, which is a vector-matrix multiplication where the elements are large polynomials of order N. Bootstrapping is therefore dominated by the calculation of the polynomial multiplications.

A schoolbook approach to polynomial multiplication would result in a computational complexity $O(N^2)$. However, utilizing the convolution theorem, the FFT can be used to compute these polynomial multiplications in time $O(N \log(N))$, as the multiplication of polynomials modulo $X^N - 1$ corresponds to a cyclic convolution of the input vectors. FHE schemes, however, need polynomial multiplications modulo $X^N + 1$, requiring *negacyclic* FFTs to compute negative-wrapped convolutions. This negacyclic convolution has a period $2N$, and thus a straightforward implementation would require $2N$ size FFTs. The cost of the negacyclic FFT on real input data can be reduced using two techniques.

The fact that the FFT computes on complex numbers offers the first opportunity for optimization. Since the input polynomials are purely real and have an imaginary component equal to zero, real-to-complex (r2c) optimized FFTs can be used, which achieves roughly a factor of two improvements in speed and memory usage [21]. This is the approach taken by the TFHE and FHEW software libraries, which compute size-2N r2c FFTs.

A second possible optimization is that negacyclic FFTs, which would have a period and size of $2N$, can be computed instead as

a regular FFT with period and size N by using a "twisting" preprocessing step [2]. During twisting, the coefficients of the input polynomial a are multiplied with the powers of the $2N$-th root of unity $\psi = \omega_{2N}$,

$$\hat{a} = (a[0], \psi a[1], \ldots, \psi^{N-1} a[N-1]). \tag{5}$$

After twisting, one can perform multiplication using a regular cyclic FFT on $\hat{a}$, halving the required FFT size to N.

While both optimizations are well-known individually, it is less straightforward to combine them. Intuitively, the twisting step is incompatible with the r2c optimization, because it will make the polynomial complex.

We use a third, but not-so-well-known technique from NuFHE [45] based on the tangent FFT [6]. The crux of this method is to "fold" polynomial coefficients $a[i]$ and $a[i+N/2]$ into a complex number $a[i] + ja[i+N/2]$ before applying the twisting step and subsequent cyclic size-$N/2$ FFT. This quarters the size of the FFT required from the original naive size-$2N$ FFT. We adopt this technique in FPT and use FFTs of size $N/2 = 256$ and $N/2 = 512$ for Parameters Sets I and II (Table 1), respectively.

3 FPT MICROARCHITECTURE

In this section, we discuss FPT's microarchitecture. First, we describe how FPT's architecture is designed as a streaming processor targeting maximum throughput. Next, we detail a batch bootstrapping technique, which significantly reduces FPT's on-chip caches and off-chip bandwidth. Finally, we present balanced implementations of the various computational stages, which enable 100% utilization of the arithmetic units during FPT's bootstrapping operation.

3.1 Streaming Processor

FHE accelerators for second-generation schemes have mostly been built after a classical CPU architecture [25, 36, 52]. They include a control unit that executes an instruction set, together with a set of arithmetic Processing Elements (PEs) that support different operations, e.g. ciphertext multiplication, key-switching, or bootstrapping. Different operations utilize different PEs, requiring careful profiling of FHE programs to balance PE relative throughputs and utilization [37, 53].

These accelerators are often memory-bound during bootstrapping, and in order to keep a high utilization level of PEs, an increasing focus is spent on optimizing the memory hierarchy, often including a multi-layer on-chip memory hierarchy with a large ciphertext register file at the lowest level.

FPT challenges this established classical CPU approach to FHE bootstrapping acceleration, and instead adopts a microarchitecture that is inspired by streaming Digital Signal Processors (DSPs). Data flows naturally through FPT's wide and directly cascaded computational stages, with simplified hard-wired routing paths and without complicated control logic. During FPT's bootstrapping operation, utilization of arithmetic units is 100%.

As illustrated in Fig. 2, FPT defines only a single fixed PE, the CMUX PE, and instantiates only a single instance of this PE with wide datapaths and massive throughput. Taking advantage of the regular structure of TFHE's PBS, consisting of n repeated CMUX iterations, this single high-throughput PE suffices to run PBS to completion. The CMUX PE computes a single PBS CMUX iteration, after which its datapath output hard-wires back into its datapath input.

Internally, the CMUX PE computes a fixed sequence of monomial multiplication, gadget decomposition, and polynomial multiply-add operations of the external product. Rather than dividing the CMUX into sub-PEs that are sequenced to run from a register file, FPT builds the CMUX with directly cascaded computational stages. Stages are throughput-balanced in the most conceivably simple way: each stage operates at the same throughput and processes a number of polynomial coefficients per clock cycle that we call the *streaming width*. Stages are interconnected in a simple fixed pipeline with static latency, avoiding complicated control logic and simplifying routing paths.

FPT is built to achieve maximum PBS throughput. As a general trend that we will detail later (Fig. 3b), the Throughput/Area (TP/A) of computational stages increases together with the streaming width. This motivates FPT to instantiate only a single wide CMUX PE with high streaming width, as opposed to many CMUX PEs with smaller streaming widths.

In summary, FPT's CMUX architecture enables massive PBS throughput by more closely resembling the architecture of a streaming Digital Signal Processor (DSP), rather than the classical CPU architecture employed by prior FHE processors.

3.2 Batch Bootstrapping

TFHE bootstrapping requires two major inputs: the input ciphertext coefficients $a_1, \ldots, a_n$ and the bootstrapping keys $BK_1, \ldots, BK_n$. Each iteration of the CMUX PE requires one element of both. The ciphertext coefficients a_i are relatively small in size and are therefore easy to accommodate. In contrast, a bootstrapping key coefficient $BK_i \in \mathsf{T}_N[X]^{(k+1)l \times (k+1)}$ is a large matrix of up to tens of kBs. Since the full BK is typically too large to fit entirely on-chip, the BK_i must be loaded from off-chip memory for every iteration. However, at high CMUX throughput levels, the required bandwidth for BK_i could easily exceed 1.0 TB/s. This is larger even than the bandwidth of HBM, and thus poses a memory bottleneck.

We propose a method, termed *batch bootstrapping*, to amortize loading the bootstrapping key for each iteration. The result is that FPT can operate entirely compute-bound, with modest off-chip bandwidth and small on-chip caches. In contrast, prior FHE processors that supported bootstrapping of second-generation schemes were often bottlenecked by the required memory bandwidth [37, 52]. In fact, a recent architectural analysis of bootstrapping [16] found that it exhibits low arithmetic intensity and requires large caches. Their conclusion was that FHE processors only benefit marginally from bespoke high-throughput arithmetic units. With our design, we show that the situation can be very different for TFHE's PBS.

In FPT, we solve the memory bottleneck problem as follows: First, due to internal pipelining, the latency of the CMUX will be much larger than its throughput. To operate at peak throughput, FPT processes multiple ciphertexts to keep its CMUX pipeline stages full. Next, we enforce that the different ciphertexts processed concurrently in the CMUX's pipeline stages arrive in a single batch of size b, encrypted under the same BK. This ensures that these

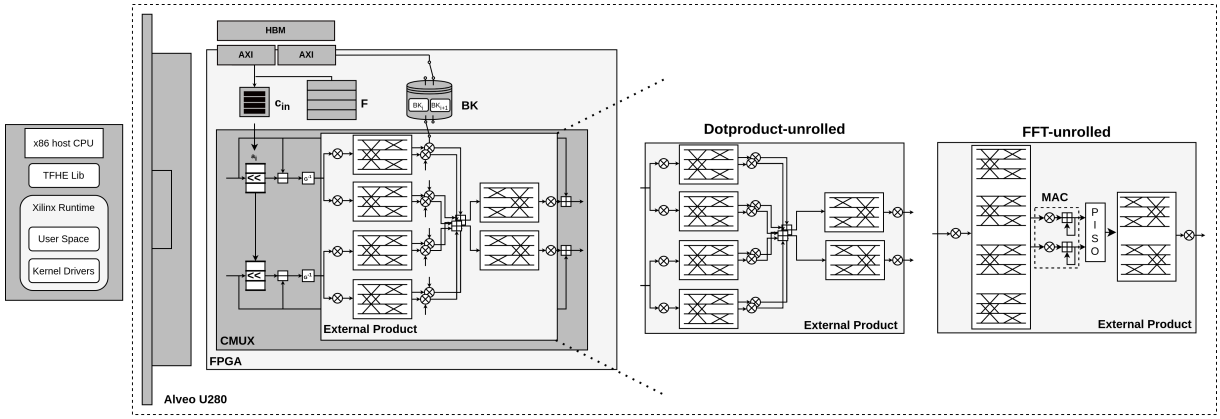


Figure 2: FPT's microarchitecture. FPT instantiates only a single PE, the CMUX PE. The CMUX is built with wide, directly cascaded datapaths, targeting massive throughput. In light grey are illustrated two throughput-balanced architectures for the external product (with $k = 1$, $l = 2$): dotproduct-unrolled (left) and FFT-unrolled (right). Host-FPGA communication includes three different interfaces: an input ciphertext FIFO, a ping-pong bootstrapping-key buffer, and a test polynomial F SRAM.

ciphertexts are at the same CMUX iteration, and as a result, all require the exact same input coefficient BK_i.

Batch bootstrapping then proceeds as follows: We instantiate a simple BRAM ping-pong buffer that holds two coefficients of BK. The CMUX reads BK_i from one half with the required bandwidth of 1.0 TB/s, while the off-chip memory fills BK_{i+1} inside the other half with a bandwidth of $1.0/b$ TB/s. In a technique similar to C-slow retiming [40], we can arbitrarily increase the batch size b by introducing more pipeline registers within the CMUX, without throughput penalty. With a batch size of $b = 16$, already the required bandwidth can be supplied by DDR4 instead of HBM.

Our simple but crucial batch bootstrapping technique exploits locality of reference to decouple the on-chip bandwidth from the off-chip bandwidth. As a result, in our architecture, TFHE's PBS is entirely compute-bound with only kB-size caches, not larger than the size of two coefficients of the bootstrapping key.

3.3 Balancing the External Product

The external product ( ), computing a vector-matrix negacyclic polynomial product, represents the bulk of the CMUX logic. As discussed before, the polynomial multiplications are performed using an FFT, and thus the ( ) operations include forward and inverse negacyclic FFT computations, and pointwise dot-products with BK_i (the bootstrapping key BK_i is already in the FFT domain).

In a streaming architecture, it is important to balance throughputs of processing elements, which is not trivial as the external product includes $(k + 1)l$ forward FFTs, but only $(k + 1)$ inverse FFT operations. We explore two different throughput-balanced architectures for the external product as shown in light-grey in Fig. 2: a dotproduct-unrolled architecture (left) and an FFT-unrolled architecture (right).

The dotproduct-unrolled architecture (left) represents the more obvious choice for parallelism, where we instantiate l times more FFT kernels compared to IFFT kernels. With the FFT-unrolled architecture on the right, we make a more unconventional choice: we balance throughputs by instantiating the FFT with l times the streaming width of the IFFT. These two architectural trade-offs can be understood as exploiting different types of "loop unrolling" inside the external product. On the left, we first loop-unroll the dot-product before unrolling the FFT, while on the right, we loop-unroll the FFT maximally.

The drawback of the FFT-unrolled architecture is that it is more complex than the dotproduct-unrolled one. First, multiply-add operations must be replaced by MACs, since polynomial coefficients that must be added are now spaced temporally over different clock cycles. Second, the inverse FFT can only start processing once a full MAC has been completed, requiring a Parallel-In Serial-Out (PISO) block that double-buffers the MAC output and matches throughputs. Third and most importantly, FFT blocks can be challenging to unroll and implement for arbitrary throughputs, and supporting two FFT blocks with differing throughputs requires non-negligible extra engineering effort.

The main advantage of the more unconventional FFT-unrolled architecture is that it features fewer FFT kernels that can therefore utilize higher streaming widths. As we will detail in the next section, this favors the general (and often-neglected) trend of pipelined FFTs, which typically feature significantly higher TP/A as the streaming width increases. At the most extreme end, a fully parallel FFT is a circuit with only constant multiplications and fixed routing paths, featuring up to 300% more throughput per DSP or per LUT on our target FPGA (Fig. 3b). FPT alleviates the extra engineering effort and extra complexity of the FFT-unrolled architecture, by extending and optimizing an existing FFT generator tool to support negacyclic FFTs.

3.4 Streaming Negacyclic FFTs

State-of-the-art FHE processors have implemented mostly *iterative* FFTs or NTTs that process polynomials in multiple passes [1, 25, 41, 49]. In these architectures, it can be difficult to support arbitrary throughputs, as memory conflicts arise when each pass requires data at different strides. Instead, FPT instantiates *pipelined* FFTs that naturally support a streaming architecture. Pipelined FFT

architectures consist of $log(N)$ stages that are connected in series. The main advantage of these architectures is that they process a continuous flow of data, which lends itself well to a fully streaming external product design.

There are many pipelined FFT architectures that target high-throughput and support arbitrary streaming widths, and we refer to [22] for a recent survey. Generally, pipelined FFTs cascade two types of units: first, the well-known butterflies with complex twiddle factor multipliers, and, second, shuffling circuits that compute stride permutations. Pipelined FFTs feature a large design space, with different possible overall architectures, area/precision trade-offs in computing twiddle factor "rotations", varying radix structures that determine which twiddle factors appear at which stages, and more. As such, they are an excellent target for tool-generated circuits, and we follow this approach for FPT.

Several FFT generator tools have been proposed in the literature. Some IP cores do not offer the massive parallelism and arbitrary streaming widths that we target for FPT [30, 61]. At the other end of the spectrum, a recent generator [24] built on top of FloPoCo [17] can only generate fully-parallel FFTs, instead of supporting arbitrary streaming widths. We synthesized at different streaming widths the High-Level Synthesis (HLS) Super Sample Rate (SSR) FFTs included in the Vitis DSP libraries of Xilinx [60], but found that they are outperformed by the RTL Verilog FFTs generated by the Spiral FFT IP Core generator [43]. Unfortunately, Spiral is not open-source and offers only a web interface towards its generated RTL [42].

Eventually, we settled on SGen[54–56] as the FFT generator tool that provided the necessary configurability, extensibility, and performance we targeted for FPT. SGen is an open-source generator implemented in Scala and employs concepts introduced in Spiral. It generates arbitrary-streaming-width FFTs through four Intermediate Representations (IRs) with different levels of optimization: an algorithm-level representation SPL, a streaming-block-level representation Streaming-SPL, an acyclic streaming IR, and an RTL-level IR. Apart from the streaming width, SGen features a configurable FFT point size, radix, and hardware arithmetic representations such as fixed-point, IEEE754 floating-point, or FloPoCo floating-point.

Most importantly, SGen is fully open-source and extensible, which we make heavy use of to generate streaming FFTs for FPT. First, we have extended SGen with operators for the forward and inverse twisting step, necessary to support negacyclic FFTs (Section 2.3). Next, we have implemented a set of optimizations aimed at higher precision and better TP/A. In this category, first, we have extended SGen with radix-2^k structures [23, 32], finding that radix-2^4 FFTs are on average 10% smaller than SGen-generated radix-4 or radix-16 FFTs. Second, we replace schoolbook complex multiplication in SGen, requiring 4 real multiplies and 2 real additions, with a variant of Karatsuba multiplications that is sometimes attributed to Gauss:

$$\begin{aligned} X + jY &= (A + jB) \cdot (C + jD) \\ Z &= C(A - B) \\ X &= (C - D)B + Z \\ Y &= (C + D)A - Z \end{aligned} \tag{6}$$

By pre-computing $C - D$ and $C + D$ for the constant twiddle factors, this multiplication requires only 3 real multiplies and 3 adds, saving scarce FPGA DSP units.

Third, we decouple the twiddle bit-width from the input bit-width. On one hand, this allows us to take advantage of the asymmetric 27×18 multipliers found in FPGA DSP blocks, while at the same time, it has been found that twiddles can be quantized with approximately four fewer bits without affecting output noise [8, 14]. Finally, as data grows throughout the FFT stages, it must initially be padded with zeros to prevent overflows. We have extended SGen with a scaling schedule, that instead divides the data by two whenever the most-significant bit must grow. Since the least-significant bits have mostly accumulated noise [59], scaling increases the precision for fixed input bit-width. Adding a scaling schedule allows us, on average, to use FFTs with 2-bit smaller fixed-point intermediate variables while meeting the same precision targets, which proves crucial to efficiently map multipliers to DSP units as will be detailed later in Section 4, Fig. 4.

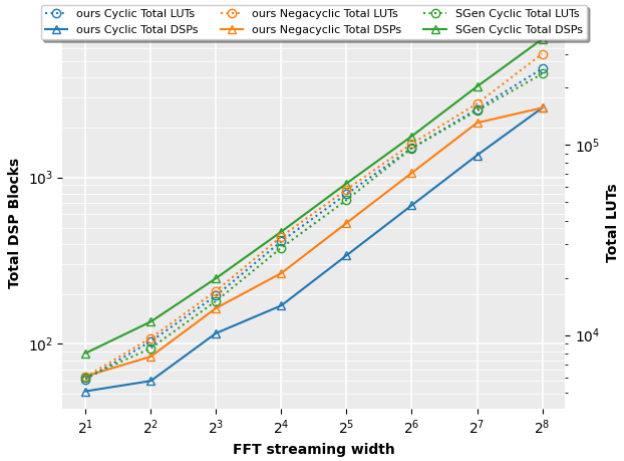
Figure 3a illustrates the resource usage of negacyclic size-256 FFTs produced by our optimized variant of SGen at different streaming widths. To quantize our improvements over SGen, we also add cyclic FFTs both with and without our introduced changes to the tool. Our changes result in significantly fewer logic resources: over 60% fewer DSP blocks are utilized while keeping LUTs comparable. As DSP blocks are the main limiting resource for FPT (Table 3), our optimizations are a key enabler to building FPT with high streaming widths.

Figure 3b illustrates our main motivation to propose the FFT-unrolled architecture for the external product. We plot the relative throughput per area unit (DSPs or LUTs) of tool-generated FFTs for different streaming widths. The trend is clear: FFTs with higher streaming widths feature up to 300% more throughput per DSP or per LUT. Intuitively, as the streaming width increases, FFTs can take more advantage of the native strengths of hardware circuits. First, shuffling circuits with MUXes and storage blocks are replaced with fixed routing paths. Second, twiddle factor multipliers can be specialized to the specific set of twiddles they need to handle, taking advantage of optimized algorithms for Single- or Multiple Constant Multiplication (SCM, MCM).

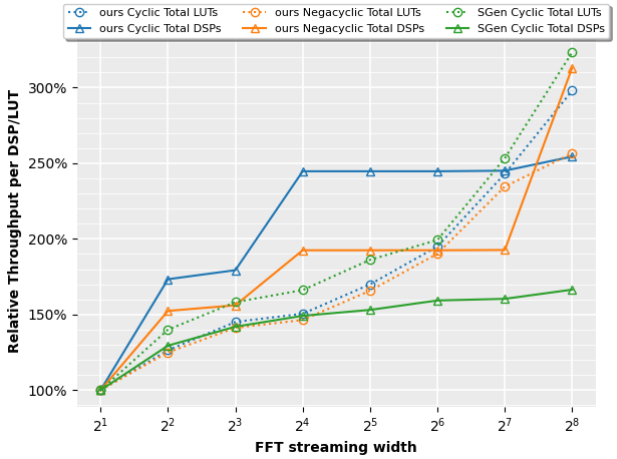
3.5 Other operations

Compared to the external product and its streaming FFTs, the remainder of the CMUX (Fig. 2, dark grey) represents mostly simple circuitry: additions, subtractions, gadget decomposition, and monomial multiplication. Whereas the first three can be streamed straightforwardly, monomial multiplication requires special treatment.

Monomial multiplication multiplies the accumulator ACC with the ciphertext-dependent monomial $X^{\lfloor \frac{2Na_i}{q} \rceil}$. Its effect is to rotate the polynomials of ACC by $\lfloor \frac{2Na_i}{q} \rceil$, and additionally negate those coefficients that wrap around. First, we truncate $\frac{2Na_i}{q}$ already in software to limit host-FPGA bandwidth. Next, an efficient architecture for monomial multiplication is a coefficient-wise barrel shifter in $log(N)$ stages.



(a)



(b)

Figure 3: FPGA resource utilization (a) and throughput / resource utilization (b) of a size-256 FFT at different streaming widths. At iso-precision, SGen FFTs use 31-bit intermediate variables without scaling schedule, and our FFTs use 29-bit intermediate variables with scaling.

To stream this operation, we define two streaming approaches: coefficient-wise streaming, and bitwise streaming. In coefficient-wise streaming, different coefficients of a polynomial are spaced temporally over different clock cycles. In bit-wise streaming, all coefficients arrive in parallel within the same clock cycle, but we instead divide different bit chunks of each coefficient over different clock cycles. One can then make a simple observation: a rotation is a difficult permutation to stream coefficient-wise, as it must interchange coefficients that are spaced over different clock cycles, but it is a simple operation to stream bit-wise, as we must simply rotate all the individual bit-chunks. We therefore add stream-reordering blocks that switch a polynomial from coefficient-wise streaming to bit-wise streaming and vice versa. At the same time, we merge the stream-reordering with the folding operation of the negacyclic FFT, which packs coefficients $a[i]$ and $a[i + N/2]$. The reordering block can be implemented at full throughput either in a R/W memory block or with a simple series of registers and MUXes.

Signed gadget decomposition involves taking unsigned 32-bit coefficients, and decomposing them into l signed coefficients of β bits. In hardware, this involves a simple reinterpretation of the bits and conditional subtraction. We merge this logic at the output of monomial multiplication to take advantage of LUT packing. In the bit-wise streamed representation, these operations must track the propagating carries in flipflops.

Gadget decomposition is approximate, e.g., for Parameter Set I, $l \cdot \beta$ = 16-bit $\ddot{Y}$ 32-bit. Contrarily to software implementations, FPT employs a CMUX datapath that is natively adjusted to approximate gadget decomposition. We discard bits prematurely that would later be rounded, allowing us to stick to a native 16-bit datapath, rather than growing back to 32-bits outside of the external product.

4 COMPACT FIXED-POINT REPRESENTATION

FFT calculations involve irrational (complex) numbers, and approximation errors arise when those numbers are represented with finite precision during computation. However, if enough precision is used, implementations of TFHE tolerate these approximation errors. More specifically, one typically aims for the total approximation error to be lower than the noise inherently present in the FHE calculations.

On a CPU, the typical method is to use floating-point numbers with single or double precision. This is efficient due to the integration of an existing Floating-Point Unit (FPU), and therefore the typical representation of choice for software designers. CPU and GPU implementations of TFHE have been restricted to double-precision floating-point FFTs because single-precision FFTs were found to introduce too much noise to guarantee successful decryption of bootstrapped ciphertexts [11].

In dedicated hardware implementations, FPUs are not natively available and are costly to include. To simplify the implementation and the analysis of the approximation error, some prior implementations opted to change the scheme to work with a prime modulus instead of a power-of-two modulus [45, 62], allowing the use of exact NTTs instead of approximate FFTs for polynomial multiplication. The downside of this approach is that one needs to include costly modular reduction units.

FPT is the first TFHE accelerator to instead utilize fixed-point calculations, which avoids the costly implementation of FPUs or modular reduction units. Moreover, instead of initializing very large fixed-point calculations to guarantee sufficient accuracy, we conduct an in-depth analysis that optimizes the fixed-point bitwidth to be just large enough so that the approximation noise is smaller than the inherent TFHE noise. FPT's optimized approach in which there is no need for a costly FPU or modular reduction unit allows a more lean and efficient design, coming at the cost of a one-time engineering effort to find the optimal parameters.

The potential effect of our fixed-point analysis on the area usage of our implementation is illustrated in Figure 4. In this figure, we plotted the LUT and DSP usage of a size-256 FFT, in function of

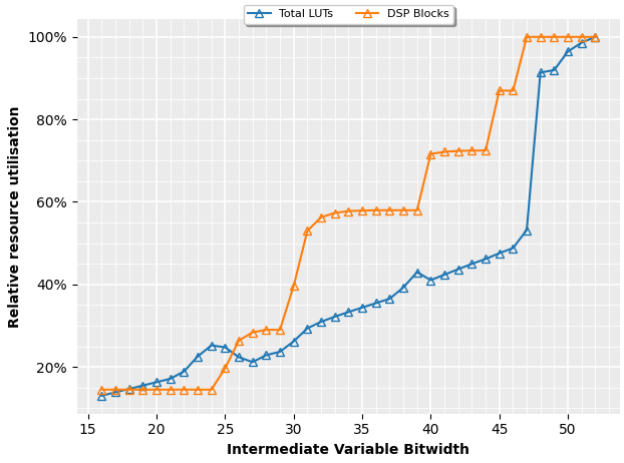


Figure 4: FPGA Relative LUT and DSP utilization of a size-256 FFT for various intermediate variable bitwidths.

the bit width of the intermediate variables. The plot gives relative numbers compared to the resource use at bitwidth 53 (loosely corresponding to the significand precision of IEEE 754 double-precision floating-point). As illustrated, reducing the bitwidth of the intermediate variables can result in a large reduction of the resource utilization, with only 20% of the LUT and DSP usage for bitwidths below 24.

Reduction of the bitwidth of intermediate variables relies on two parts, the location of the most significant bit, and the location of the least significant bit. We will first look at our strategy to set the MSB position of intermediate variables, and then focus on the LSB.

4.1 Setting the MSBs

The location of the most significant bit is important to avoid overflows. If an overflow occurs, the intermediate variable will be completely distorted and thus the result of the calculation will be unusable. Two strategies can be adopted to deal with overflows: a worst-case scenario where one can choose parameters to avoid any overflow or an average-case scenario where one allows a certain overflow with sufficiently low probability.

Avoiding any overflow comes at a significant enlargement of the parameters and thus at a significant cost, which is why we adopt the strategy to avoid overflows with a maximal overflow probability of $P_{of} = 2^{-64}$. To determine the ideal MSB position we measure the variance and then assume a Gaussian distribution to calculate the overflow probability. For a given MSB position p_{MSB} and standard deviation σ, the probability of overflow is:

$$P_{of} = P[|\chi| \; i \; 2^{p_{MSB}}/2 \,|\, \chi \xleftarrow{\$} \mathcal{N}(0,\sigma)] \tag{7}$$

$$= 1 - \text{erf}\left(\frac{2^{p_{MSB}}}{2\sqrt{2}\sigma}\right). \tag{8}$$

Using this equation we determine the lowest p_{MSB} that fulfills the maximal overflow probability of $P_{of} = 2^{-64}$ for each intermediate variable.

Parameter Set	(I)	(II)
BK	$\texttt{FixedPoint}_{26}\texttt{(7, 19)}$	$\texttt{FixedPoint}_{27}\texttt{(8, 19)}$
FFT	$\texttt{FixedPoint}_{29}\texttt{(15, 14)}$	$\texttt{FixedPoint}_{30}\texttt{(18, 12)}$
IFFT	$\texttt{FixedPoint}_{29}\texttt{(23, 6)}$	$\texttt{FixedPoint}_{30}\texttt{(27, 3)}$

Table 2: Fixed-point data representations used by intermediate variables, in the format $\texttt{FixedPoint}_{\texttt{width}}\texttt{(integerBits, fractionalBits)}$.

4.2 Setting the LSBs

The position of the least significant bits has an influence on the approximation noise that is introduced during the calculations. This approximation noise can be tolerated up to a certain level. More specifically, the approximation noise should be small enough so that the combination of the approximation noise and the inherent TFHE noise still leads to a correct bootstrap with high probability. We divide the total acceptable noise, for which we use theoretical noise bounds of [11], into two equal parts for the approximation noise and the inherent noise, thus allowing our approximation noise to be at most half the total acceptable noise.

In our design, we focus on three main parameters: the intermediate variable widths during the forward and inverse FFT calculations, and the bitwidth of the coefficients of the bootstrapping key BK. We assume the noise introduced due to each parameter is independent (as each parameter comes from a separate block in our design), which means that the variance of the total noise is equal to the sum of the variances of each noise source ($\sigma^2_{tot} = \sigma^2_{FFT} + \sigma^2_{IFFT} + \sigma^2_{BK}$). We then limit the noise variance due to each noise source to $1/3$th of the total noise variance.

To find optimal fixed-point parameter values, we perform a parameter sweep by setting the parameters to very high widths (in our example 53) resulting in very low noise, and then iteratively reducing one parameter until it hits the noise threshold while keeping the other parameters at high widths. The result of this experiment can be seen in Fig. 5, and our final fixed-point parameters are illustrated in Table 2. Note that we give the IFFT data representation before outputs are scaled by $1/N$.

4.3 Related and Future Work

One prior implementation proposing a custom hardware format for TFHE's FFTs is MATCHA [33], who propose to use (integer) Dyadic-Value-Quantized Twiddle Factors (DVQTFs). Our fixed-point parameter analysis improves on MATCHA's in two key ways:

First, MATCHA only considers the bitwidth of twiddle factors, and set a uniform bitwidth (either 38-bit or 64-bit) that is employed throughout their external product calculations. Our analysis instead shows that different intermediate variables can profit from different fixed-point representations, allowing for an overal smaller resource utilisation (Fig. 5, Table 2). Moreover, our analysis allows us to quantize BK smaller than other parameters, limiting both on-chip BK_i buffers and off-chip bandwidth.

Second, in MATCHA, instead of measuring the noise variance, the authors conduct 10^8 tests for a parameter set to test if there are no decryption failures at the end of bootstrapping. The downside of this approach is that it becomes expensive when multiple

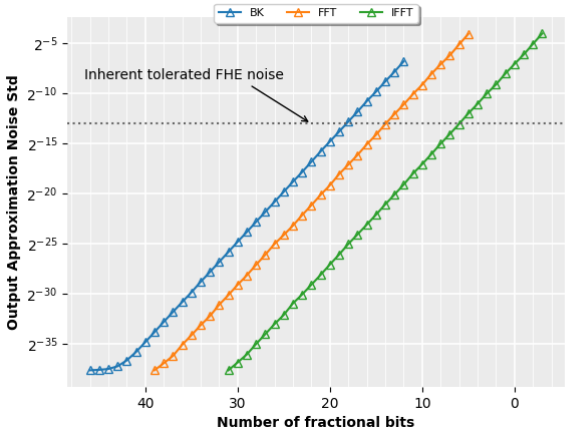


Figure 5: Output approximation noise versus the number of fractional bits for the representation of the bootstrapping key and intermediate variables during the forward and inverse FFT.

parameters have to be set. Furthermore, this methodology does not give exact values of the failure probability, as one only has the information that no errors were found in 10^8 tests. Our approach of measuring the approximation noise and matching it with the theoretical noise bounds provides for a more rigorous and lean design.

Finally, we note that there are other intermediate variables that could be optimized, for example, the widths of the twiddle factors in the FFT calculations. We heuristically set them to the width of the intermediate variables minus 4, which gave a good balance between failure probability and cost as also explained in Section 3.4. Interesting future work could include a full search over all possible parameters, which could result in improved fixed-point parameters over our heuristic approach.

5 IMPLEMENTATION

We implemented FPT for a Xilinx Alveo U280 datacenter accelerator FPGA featuring 1.3M LUTs, 2.6M FFs, 9024 DSPs, and 41 MB of on-chip SRAM. For both parameter sets, we employ our FFT-unrolled architecture with a forward FFT streaming width of 128 complex coefficients/clock cycle, and an IFFT streaming width of $128/l = 64$ complex coefficients per clock cycle. For Parameter Set II with $N = 1024$, we have also implemented the dotproduct-unrolled architecture with $(k+1)l = 4$ forward FFT kernels and $(k+2) = 2$ IFFT kernels, both of uniform streaming width 32. At this datapoint, providing iso-throughput to the FFT-unrolled architecture, we found that the dotproduct-unrolled architecture incurs 10% more average DSP and LUT usage, and we therefore do not evaluate it further.

Our FFT-unrolled architectures feature massive throughput, completing one CMUX every $(256/128) \cdot (k+1)l = 12$ clock cycles for Parameter Set I, and every $(512/128) \cdot (k+1)l = 16$ clock cycles for Parameter Set II. The latency of the CMUX is larger: 156 cycles for Parameter Set I, and 224 cycles for Parameter Set II. In both cases, we operate at peak throughput by filling the CMUX pipeline with a batch of ciphertexts, of sizes $b = 156/12 = 13$ and $b = 224/16 = 14$, respectively.

5.1 External I/O

The Alveo U280 includes three different host-FPGA memory interfaces: 32 GB of DDR4, 8 GB of HBM accessed through 32 Pseudo-Channels (PCs), and 24 MB of PLRAM. PBS also requires three host-side inputs: a batch of b input ciphertexts c_{in}, the long-term bootstrapping key BK, and the test polynomial LUT F to evaluate over the ciphertext.

For the long-term bootstrapping key, we note that it is not absolutely necessary to instantiate a ping-pong BK_i buffer, as discussed in Section 3.2, on our target Alveo U280 FPGA. For our parameter sets and fixed-point-trimmed BK bitwidths, the full BK measures approximately 15 MB and fits entirely in a combination of the on-chip BRAM and URAM. Nevertheless, we instantiate a small ping-pong BK_i cache as a proof-of-concept. This requires an on-chip ping-pong buffer of only $2/n$ of the full size of BK, allowing our architecture to remain compute-bound on architectures with less on-chip SRAM, such as smaller FPGAs or heavily memory-trimmed down ASICs. Moreover, our technique ensures that our architecture scales to new TFHE algorithms or related schemes like FHEW that increase the size of the bootstrapping key.

For our batch sizes b, the required BK bandwidth is only tens of GB/s, which we provide by splitting the BK over a limited number of HBM PCs 0-7, each providing 14 GB/s of bandwidth. The input and output ciphertext batches are small and require negligible bandwidth, which we allocate in a single HBM PC. Each HBM channel is served by a separate AXI master on the PL-side, which are R/W for the ciphertext and read-only for BK. For the test polynomial LUT F, we allocate an on-chip RAM that can store a configurable number of test polynomials. Each input ciphertext is tagged with an index of the LUT to apply, and correspondingly the test polynomial F to select from the RAM as input to the first CMUX iteration. LUTs depend on the specific FHE program, are typically limited in number, and do not change often. For example, bootstrapped Boolean gates require only a single LUT. As such, we keep the RAM small, and we share the same HBM PC and AXI master that is used by the input and output ciphertexts.

5.2 Xilinx Run Time Kernel

FPT is accessible from the host as Xilinx Run Time (XRT) RTL kernel and managed through XRT API calls. FPT's CMUX pipeline features 100% utilization during a single ciphertext batch bootstrap, and does not require complex kernel overlapping to reach peak throughput. To ensure that there are no pipeline bubbles *between* the bootstrapping of different batches, we allow early pre-fetching of the next ciphertext batch into an on-chip FIFO. As such, we build FPT to support the Vitis `ap_ctrl_chain` kernel block level control protocol, which permits overlapping kernel executions and allows FPT to queue future ciphertext batch base HBM addresses.

	LUT	FF	DSP	BRAM
	(40% av.)	(35% av.)	(61% av.)	(25% av.)
FPT (I)	526K	916K	5494	505
CMUX	384K	707K	5494	310
MAC (384×)	97K	114K	2304	310
$FFT_{256,128}$	159K	366K	2126	0
$IFFT_{256,64}$	97K	192K	1064	0
	(46% av.)	(39% av.)	(66% av.)	(20% av.)
FPT (II)	595K	1024K	5980	412
CMUX	458K	827K	5980	215
MAC (256×)	66K	79K	1536	215
$FFT_{512,128}$	222K	449K	2958	0
$IFFT_{512,64}$	130K	255K	1486	0

Table 3: FPT Hardware Resource Utilization Breakdown for Parameter Sets I and II. DSP blocks are the main limiting resource with up to 66% of available FPGA resources utilized by FPT.

5.3 Fixed-point Streaming Design in Chisel

While the outer host-FPGA communication logic of FPT is implemented in SystemVerilog, we use Chisel [4] – an open-source HDL embedded in Scala – to construct the inner streaming CMUX kernel. Like SystemVerilog, Chisel is a full-fledged HDL with direct constructs to describe synthesizable combinational and sequential logic and not a High-Level Synthesis (HLS) language. Our motivation to select Chisel over SystemVerilog for the CMUX, is that it makes the full capabilities of the Scala language available to describe circuit generators. We make heavy use of object-oriented and functional programming tools to describe our CMUX streaming architecture for a configurable streaming width, and in both realizations shown in Fig. 2. Moreover, Chisel has a rich typesystem that is further supported by external libraries. In FPT, the existing `DspComplex[FixedPoint]` is our main hardware datatype that we use within our architecture. Building on existing `FixedPoint` test infrastructure that we extended for FPT, our experiments in Section 4 are directly run on the Chisel-generated Verilog rather than an intermediate fixed-point software model.

6 EVALUATION AND COMPARISON

6.1 Resource Utilization

FPT is implemented using Xilinx Vivado 2022.2 and packaged as XRT kernel using Vitis 2022.2, targeting a clock frequency of 200 MHz. Table 3 presents a resource utilization breakdown of FPT, for both Parameter Sets I and II. In both cases, DSP blocks are the main limiting resource that prevents increasing to the next available streaming width, with up to 66% of available DSP blocks utilized by FPT. Note that whereas Fig. 2 presented our ping-pong *BK* buffer as a monolithic memory block, it is physically split into many smaller memory blocks that are placed inside the MAC units that consume them.

6.2 PBS Benchmarks

Table 4 compares FPT quantitatively with a number of prior TFHE baselines. For our CPU baseline, we benchmark single-core PBS in CONCRETE[12] on an Intel Xeon Silver 4208 CPU at 2.1 GHz. A recent ASIC baseline is provided by MATCHA [33]. MATCHA presents emulations of a 36.96mm^2 ASIC in a 16nm PTM process technology. As FPGA baseline, we include a recent architecture of Ye et al. [62], which was developed concurrently with our work and significantly improves the prior baseline of Gener et al. [26]. We refer to this architecture by the author initials YKP, and we also include YKP's benchmarks of cuFHE [15], a GPU-based implementation benchmarked on an NVIDIA GeForce RTX 3090 GPU at 1.7 GHz, in our comparison.

The main design goal of FPT is PBS throughput. Table 4 illustrates the massive PBS throughput that is enabled through FPT's streaming architecture: 937× more than CONCRETE, 7.1× more than YKP, and 2.5× more than MATCHA or cuFHE.

In FPT's current instantiation, we did not optimize for latency. As the PCIe and AXI latencies of communicating the in- and output ciphertext batches are negligible, FPT's PBS latency is mostly determined by its CMUX pipeline depth. In this work, we kept the CMUX pipeline depth large, fitting b ciphertexts and enabling small off-chip bandwidth through our batched bootstrapping technique. Lower-latency implementations of FPT can opt to decrease the CMUX pipeline depth, requiring either more off-chip bandwidth to load *BK* or caching the full *BK* on-chip. Nevertheless, FPT's latency even in its current instantiation is competitive with MATCHA. We note that FPT is estimated at 99W total on-chip power (FPGA and HBM), offering a similar TP/W as MATCHA (40W) and significantly more than cuFHE ($¡$ 200W) or YKP (50W).

6.3 Related Work

Qualitatively, FPT makes different design choices than either YKP or MATCHA. MATCHA is built after the classical CPU approach to FHE accelerators. It includes a set of TGGSW clusters with external product cores that operate from a register file. As one result, MATCHA is bottlenecked by data movement and cache memory access conflicts.

YKP is an HLS-based architecture that redefines TFHE to use NTT, breaking compatibility with existing TFHE libraries like CONCRETE, and disabling the fixed-point optimizations of FPT. At the architectural level, YKP includes some concepts also employed by FPT. Similar to FPT, they include a pipelined implementation of the CMUX that processes multiple ciphertext instances. However, unlike FPT which builds a single streaming CMUX PE with large and configurable streaming width, YKP implements and instantiates multiple smaller CMUX PEs with inferior TP/A. Each CMUX pipeline instance in YKP includes an SRAM that stores a coefficient BK_i. However, unlike FPT where these SRAMs are loaded by off-chip memory in ping-pong fashion, YKP loads coefficients from DRAM only after a full coefficient has been consumed. This makes the number of CMUX PEs they instantiate limited by the off-chip memory bandwidth, whereas FPT's design choices make it entirely compute-bound.

	Parameter Set	LUT / FFs / DSP / BRAM	Clock (MHz)	Latency(ms)	TP (PBS/ms)
FPT	(I)	526K / 916K / 5494 / 17.5Mb	200	0.48	**28.4**
	(II)	595K / 1024K / 5980 / 14.5Mb	200	0.58	**25.0**
YKP [62]	(II)	842K / 662K / 7202 / 338Mb	180	3.76	3.5
		442K / 342K / 6910 / 409Mb	180	1.88	2.7
MATCHA [33]	(II)	36.96mm^2 16nm PTM	2000	0.2	10
CONCRETE[12]	(I)	Intel Xeon Silver 4208	2100	33	0.03
	(II)		2100	32	0.03
cuFHE [15]	(II)	NVIDIA GeForce RTX 3090	1700	9.34	9.6

Table 4: Comparison of TFHE PBS on a variety of platforms

Both MATCHA and YKP focus on an algorithmic technique called *bootstrapping key unrolling*. This technique unrolls m iterations of the Blind Rotation loop (Algorithm 1, Line 2), requiring an (exponentially) more expensive CMUX equation and larger BK, but reducing the total number of iterations from n to n/m. Bootstrapping key unrolling was not considered for FPT, but is a promising future technique to evaluate. Moreover, since FPT is not bound by off-chip memory bandwidth, more aggressive key unrolling could be feasible.

For completeness, we note that both MATCHA and YKP include key-switching as an operation of PBS. Key-switching includes coefficient-wise multiplication of a TLWE ciphertext with a key-switching key. We opted not to include key-switching in FPT, because different FHE programs may choose to key-switch either before or after PBS [11]. Nevertheless, key-switching is an operation with much lower throughput requirements than the CMUX [62]. In FPT, key-switching of the output ciphertext can be supported without throughput penalty (but with slightly increased latency) by instantiating a few integer multipliers on the AXI write-back path.

7 CONCLUSION

In this paper, we introduced FPT, an accelerator for the Torus Fully Homomorphic Encryption (TFHE) scheme. In contrast to previous FHE architectures, our design follows a streaming approach with high throughput and low control overhead. Owing to a batched design and balanced streaming architecture, our accelerator is the first FHE bootstrapping implementation that is compute-bound and not memory-bound, with small data caches and a 100% utilization of the arithmetic units. Instead of using an NTT or floating-point FFT, FPT achieves a significant throughput increase by utilizing up to 80% area-reduced fixed-point FFTs with compact and optimized variable representations. In the end, FPT achieves a TFHE bootstrapping throughput of 28.4 bootstrappings per millisecond, which is 937× faster than CPU implementations, 7.1× faster than a concurrent FPGA implementation, and 2.5× faster than state-of-the-art ASIC and GPU designs.

ACKNOWLEDGMENTS

This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, the Research Council KU Leuven (C16/15/058), the Horizon 2020 ERC Advanced Grant (101020005 Belfort), and the AMD Xilinx University Program through the donation of a Xilinx Alveo U280 datacenter accelerator card.

Michiel Van Beirendonck is funded by FWO as Strategic Basic (SB) PhD fellow (project number 1SD5621N). Jan-Pieter D'Anvers is funded by FWO (Research Foundation – Flanders) as junior post-doctoral fellow (contract number 133185).

Finally, the authors would like to thank Wouter Legiest for experimenting with a variety of FFT generator tools, and Furkan Turan for experimenting with the Alveo U280's external I/O interfaces.

REFERENCES

[1] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2022. FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption. *CoRR* abs/2207.11872 (2022). https://doi.org/10.48550/arXiv.2207.11872 arXiv:2207.11872

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

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (2010), 50–58. https://doi.org/10.1145/1721654.1721672

[4] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun (Eds.). ACM, 1216–1225. https://doi.org/10.1145/2228360.2228584

[5] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. 2018. High-Performance FV Somewhat Homomorphic Encryption on GPUs: An Implementation using CUDA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018, 2 (2018), 70–95. https://doi.org/10.13154/tches.v2018.i2.70-95

[6] Daniel J. Bernstein. 2007. The Tangent FFT. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 17th International Symposium, AAECC-17, Bangalore, India, December 16-20, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4851)*, Serdar Boztas and Hsiao-feng Lu (Eds.). Springer, 291–300. https://doi.org/10.1007/978-3-540-77224-8_34

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3 (2014), 13:1–13:36. https://doi.org/10.1145/2633600

[8] Wei-Hsin Chang and Truong Q. Nguyen. 2008. On the Fixed-Point Accuracy Analysis of FFT Algorithms. *IEEE Trans. Signal Process.* 56, 10-1 (2008), 4673–4682. https://doi.org/10.1109/TSP.2008.924637

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10624)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10031)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). 3–33. https://doi.org/10.1007/978-3-662-53887-6_1

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *J. Cryptol.* 33, 1 (2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[12] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2020. CONCRETE: Concrete operates on ciphertexts rapidly by extending TfhE. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, Vol. 15.

[13] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 2021. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. In *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12716)*, Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann (Eds.). Springer, 1–19. https://doi.org/10.1007/978-3-030-78086-9_1

[14] Ainhoa Cortés, Igone Vélez, Ibon Zalbide, Andoni Irizar, and Juan F. Sevillano. 2008. An FFT Core for DVB-T/DVB-H Receivers. *VLSI Design* 2008 (2008), 610420:1–610420:9. https://doi.org/10.1155/2008/610420

[15] Wei Dai and Berk Sunar. 2015. cuHE: A Homomorphic Encryption Accelerator Library. In *Cryptography and Information Security in the Balkans - Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9540)*, Enes Pasalic and Lars R. Knudsen (Eds.). Springer, 169–186. https://doi.org/10.1007/978-3-319-29172-7_11

[16] Leo de Castro, Rashmi Agrawal, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. 2021. Does Fully Homomorphic Encryption Need Compute Acceleration? *CoRR* abs/2112.06396 (2021). arXiv:2112.06396 https://arxiv.org/abs/2112.06396

[17] Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Des. Test Comput.* 28, 4 (2011), 18–27. https://doi.org/10.1109/MDT.2011.44

[18] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. 2015. Accelerating Fully Homomorphic Encryption in Hardware. *IEEE Trans. Computers* 64, 6 (2015), 1509–1521. https://doi.org/10.1109/TC.2014.2345388

[19] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9056)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, 617–640. https://doi.org/10.1007/978-3-662-46800-5_24

[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* (2012), 144. http://eprint.iacr.org/2012/144

[21] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. https://doi.org/10.1109/JPROC.2004.840301

[22] Mario Garrido. 2021. A Survey on Pipelined FFT Hardware Architectures. *Journal of Signal Processing Systems* (06 Jul 2021). https://doi.org/10.1007/s11265-021-01655-1

[23] Mario Garrido, Jesús Grajal, Miguel A. Sánchez Marcos, and Oscar Gustafsson. 2013. Pipelined Radix-2^k Feedforward FFT Architectures. *IEEE Trans. Very Large Scale Integr. Syst.* 21, 1 (2013), 23–32. https://doi.org/10.1109/TVLSI.2011.2178275

[24] Mario Garrido, Konrad Möller, and Martin Kumm. 2019. World's Fastest FFT Architectures: Breaking the Barrier of 100 GS/s. *IEEE Trans. Circuits Syst. I Regul. Pap.* 66-I, 4 (2019), 1507–1516. https://doi.org/10.1109/TCSI.2018.2886626

[25] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. 2022. BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption. *CoRR* abs/2205.14017 (2022). https://doi.org/10.48550/arXiv.2205.14017 arXiv:2205.14017

[26] Serhan Gener, Parker Newton, Daniel Tan, Silas Richelson, Guy Lemieux, and Philip Brisk. 2021. An FPGA-based Programmable Vector Engine for Fast Fully Homomorphic Encryption over the Torus. In *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*.

[27] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, Michael Mitzenmacher (Ed.). ACM, 169–178. https://doi.org/10.1145/1536414.1536440

[28] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6632)*, Kenneth G. Paterson (Ed.). Springer, 129–148. https://doi.org/10.1007/978-3-642-20465-4_9

[29] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, 75–92. https://doi.org/10.1007/978-3-642-40041-4_5

[30] LLC Gisselquist Technology. [n. d.]. A Generic Piplined FFT Core Generator. https://github.com/ZipCPU/dblclockfft.

[31] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. 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*. AAAI Press, 9466–9471. https://doi.org/10.1609/aaai.v33i01.33019466

[32] Shousheng He and Mats Torkelson. 1996. A New Approach to Pipeline FFT Processor. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, 1996, Honolulu, Hawaii, USA*. IEEE Computer Society, 766–770. https://doi.org/10.1109/IPPS.1996.508145

[33] Lei Jiang, Qian Lou, and Nrushad Joshi. 2022. MATCHA: a fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, Rob Oshana (Ed.). ACM, 235–240. https://doi.org/10.1145/3489517.3530435

[34] Marc Joye. 2022. SoK: Fully Homomorphic Encryption over the [Discretized] Torus. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 4 (2022), 661–692. https://doi.org/10.46586/tches.v2022.i4.661-692

[35] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 4 (2021), 114–148. https://doi.org/10.46586/tches.v2021.i4.114-148

[36] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 1237–1254. https://doi.org/10.1109/MICRO56248.2022.00086

[37] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 711–725. https://doi.org/10.1145/3470496.3527415

[38] Igor Kononenko. 2001. Machine learning for medical diagnosis: history, state of the art and perspective. *Artif. Intell. Medicine* 23, 1 (2001), 89–109. https://doi.org/10.1016/S0933-3657(01)00077-X

[39] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022. Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network. *IEEE Access* 10 (2022), 30039–30054. https://doi.org/10.1109/ACCESS.2022.3159694

[40] Charles E. Leiserson and James B. Saxe. 1991. Retiming Synchronous Circuitry. *Algorithmica* 6, 1 (1991), 5–35. https://doi.org/10.1007/BF01759032

[41] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. 2022. Medha: Microcoded Hardware Accelerator for computing on Encrypted Data. *CoRR* abs/2210.05476 (2022). https://doi.org/10.48550/arXiv.2210.05476 arXiv:2210.05476

[42] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. [n. d.]. Spiral DFT/FFT IP Core Generator. https://www.spiral.net/hardware/dftgen.html.

[43] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2012. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Trans. Design Autom. Electr. Syst.* 17, 2 (2012), 15:1–15:33. https://doi.org/10.1145/2159542.2159547

[44] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, Mizan Abraha Gebremichael, Homer Gamil, Eduardo Chielle, Ramesh Karri, Mihai Sanduleanu, and Michail Maniatakos. 2022. CoFHEE: A Co-processor for Fully Homomorphic Encryption Execution. *CoRR* abs/2204.08742 (2022). https://doi.org/10.48550/arXiv.2204.08742 arXiv:2204.08742

[45] NuCypher. [n. d.]. NuFHE, a GPU-powered Torus FHE implementation. https://github.com/nucypher/nufhe/.

[46] Nicolas Papernot, Patrick D. McDaniel, Arunesh Sinha, and Michael P. Wellman. 2018. SoK: Security and Privacy in Machine Learning. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 399–414. https://doi.org/10.1109/EuroSP.2018.00035

[47] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrián Macías. 2015. Accelerating Homomorphic Evaluation on Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9293)*, Tim Güneysu and Helena Handschuh (Eds.). Springer, 143–163. https://doi.org/10.1007/978-3-662-48324-4_8

[48] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria E. Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. 2019. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *ACM Comput. Surv.* 51, 5 (2019), 92:1–92:36. https://doi.org/10.1145/3234150

[49] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1295–1309. https://doi.org/10.1145/3373376.3378523

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

[51] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 387–398. https://doi.org/10.1109/HPCA.2019.00052

[52] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 238–252. https://doi.org/10.1145/3466752.3480070

[53] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 173–187. https://doi.org/10.1145/3470496.3527393

[54] François Serre and Markus Püschel. 2018. A DSL-Based FFT Hardware Generator in Scala. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 315–322. https://doi.org/10.1109/FPL.2018.00060

[55] François Serre and Markus Püschel. 2019. DSL-Based Modular IP Core Generators: Example FFT and Related Structures. In *26th IEEE Symposium on Computer Arithmetic, ARITH 2019, Kyoto, Japan, June 10-12, 2019*, Naofumi Takagi, Sylvie Boldo, and Martin Langhammer (Eds.). IEEE, 190–191. https://doi.org/10.1109/ARITH.2019.00043

[56] François Serre and Markus Püschel. 2020. DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks. *ACM Trans. Recon gurable Technol. Syst.* 13, 1 (2020), 1:1–1:23. https://doi.org/10.1145/3359754

[57] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA. *IEEE Trans. Computers* 69, 8 (2020), 1185–1196. https://doi.org/10.1109/TC.2020.2988765

[58] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2012. Accelerating fully homomorphic encryption using GPU. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*. IEEE, 1–5. https://doi.org/10.1109/HPEC.2012.6408660

[59] P. Welch. 1969. A fixed-point fast Fourier transform error analysis. *IEEE Transactions on Audio and Electroacoustics* 17, 2 (1969), 151–157. https://doi.org/10.1109/TAU.1969.1162035

[60] Xilinx. [n. d.]. Vitis DSP Library. https://xilinx.github.io/Vitis_Libraries/dsp.

[61] Xilinx. 2022. Fast Fourier Transform v9.1. LogiCORE IP Product Guide. PG109.

[62] Tian Ye, Rajgopal Kannan, and Viktor K. Prasanna. 2022. FPGA Acceleration of Fully Homomorphic Encryption over the Torus. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*. IEEE, 1–7. https://doi.org/10.1109/HPEC55821.2022.9926381

[63] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. 2020. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access* 8 (2020), 58443–58469. https://doi.org/10.1109/ACCESS.2020.2983149

[64] Zama. 2022. Announcing Concrete-core v1.0.0-gamma with GPU acceleration. https://www.zama.ai/post/concrete-core-v1-0-gamma-with-gpu-acceleration