Hardware-Accelerated Encrypted Execution of General-Purpose Applications

Charles Gouert[1], Vinu Joseph[2], Steven Dalton[2],
Cedric Augonnet[2], Michael Garland[2] and Nektarios Georgios Tsoutsos[1]

[1] University of Delaware
[2] NVIDIA

Abstract. Fully Homomorphic Encryption (FHE) is a cryptographic method that guarantees the privacy and security of user data during computation. FHE algorithms can perform unlimited arithmetic computations directly on encrypted data without decrypting it. Thus, even when processed by untrusted systems, confidential data is never exposed. In this work, we develop new techniques for accelerated encrypted execution and demonstrate the significant performance advantages of our approach. Our current focus is the Fully Homomorphic Encryption over the Torus (CGGI) scheme, which is a current state-of-the-art method for evaluating arbitrary functions in the encrypted domain. CGGI represents a computation as a graph of homomorphic logic gates and each individual bit of the plaintext is transformed into a polynomial in the encrypted domain. Arithmetic on such data becomes very expensive: operations on bits become operations on entire polynomials. Therefore, evaluating even relatively simple nonlinear functions with the CGGI cryptosystem, such as a sigmoid, can take thousands of seconds on a single CPU thread. Using our novel framework for end-to-end accelerated encrypted execution called ArctyrEX, developers with no knowledge of complex FHE libraries can simply describe their computation as a C program that is evaluated 18× faster on average relative to the GPU-accelerated Concrete library for multiplication-intensive benchmarks.

Keywords: Fully homomorphic encryption, high performance computing, GPU acceleration, data privacy.

1 Introduction

Cloud computing allows users to forego the practice of maintaining costly data centers in house, and provides both computation and storage capabilities on-demand. However, all user data will necessarily reside on servers owned by the cloud service provider who could view the uploaded data. Additionally, attackers are increasingly targeting cloud servers because they contain sensitive data from multiple users [57] [6] [64]. FHE allows users to encrypt data locally, outsource the ciphertexts to the cloud for oblivious and meaningful computation, and receive the encrypted processed data for decryption. This can be used for a wide variety of applications, such as privacy-preserving machine learning as a service (MLaaS) [29] [23] [15] and facial recognition [65] [48].

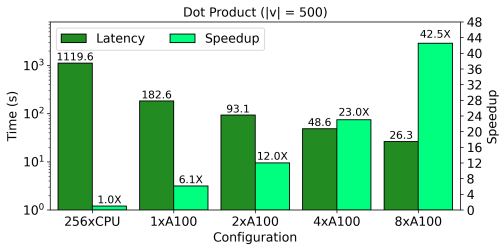
FHE was realized in 2009 with the advent of the bootstrapping procedure which allows unlimited computation on ciphertexts [30]. However, early FHE was plagued by both high memory requirements and enormous computational overheads, which rendered it infeasible for adoption. Since its inception, great strides have been made to reduce these runtime costs: First, new homomorphic encryption schemes have been developed with more efficient bootstrapping constructions, such as DM [27] and CGGI [18]. Additionally, various algorithmic and software optimizations, such as HE-friendly number theoretic

```
int dot_product(int x[500], int y[500]) {
  int product = 0;
  for (int i = 0; i < 500; i++)
    product = product + x[i] * y[i];
  return product;
}
```

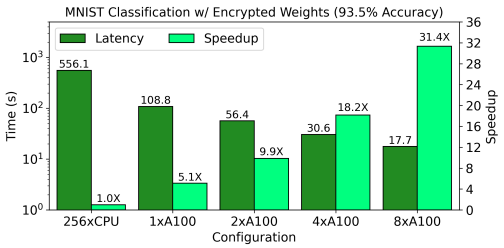
(a) Dot Product Code

```
void fc_layer(short x[256],
              short w[7680],
              short res[30]) {
  for (int i = 0; i < 30; i++)
    for (int j = 0; j < 256; j++)
        res[i] = res[i] + x[j] * w[256 * i + j];
}
```

(b) Fully-Connected Layer Code



(c) Dot Product Performance



(d) MNIST Classification Performance

Figure 1: **Demonstration Applications**: Using our approach, a dot product subroutine runs $\approx 6\times$ faster on a single A100 GPU and over $42\times$ faster with an NVIDIA DGX A100 relative to an AMD EPYC 7742 baseline (with 256 threads), resulting in an end-to-end application level speed up of $31\times$ for our demonstration using MNIST classification.

transforms (NTT) [25], have yielded significant speedups in encrypted computation for certain core operations. Additionally, utilization of the residue number system (RNS) has been employed to enhance parallelism and avoid large integer arithmetic [39] [16] [7]. Lastly, CPU-based acceleration techniques were also adopted, including AVX and FMA extensions [10]. However, the algorithmic level performance gains have begun to stagnate and further speedups are coming mostly from hardware acceleration.

Numerous hardware platforms for FHE computation show promise, such as GPUs [26, 20], ASICs [56, 47], FPGAs [54], and even optical processors (OPUs) [67]. In this work, we focus on GPUs, which have been thoroughly demonstrated to be well-suited to the primitives underlying encrypted computation with FHE. Most encrypted operations expose ample parallelism and are computationally intensive [45]; therefore, FHE applications can leverage the high degrees of parallelism afforded by these devices. For instance, a 10×10 matrix multiplication in the encrypted domain using the CGGI cryptosystem in gate bootstrapping mode [18] requires hundreds of millions of large polynomial arithmetic operations and NTTs. Open-source nuFHE [50] and cuFHE [26] libraries expose an API akin to an assembly language, requiring programmers to compose their algorithms as Boolean circuits and their goal was to maximize the performance of individual homomorphic operations, as opposed to end-to-end encrypted applications themselves.

In this work, we propose a framework called ArctyrEX (Accelerated Encrypted eXecution) demonstrate that GPU-accelerated FHE can be used to greatly improve the efficiency of realistic and representative FHE applications, such as neural network inference and large linear algebra arithmetic. We also introduce automated scheduling techniques that allow for strong scalability while evaluating encrypted algorithms with multiple GPUs. Notably, most cryptographic details and all hardware acceleration functionalities are handled automatically by ArctyrEX to minimize the burden on programmers. Our key contributions can be summarized as follows:

- A custom algorithm to translate high-level code to GPU-friendly FHE programs that reduces latency by up to 36%, while also reducing circuit size by up to 40% relative to a standard synthesis flow;

- A novel scheduling methodology that facilitates efficient computation across multiple GPUs, which enables encrypted programs to run 18× faster for multiplication-intensive benchmarks on 2 GPUs relative to the state-of-the-art Concrete library [19];

- A new CUDA-accelerated backend for the CGGI cryptosystem that prioritizes fast evaluation of arbitrary algorithms and outperforms state-of-the-art multi-threaded CPU and GPU implementations by more than an order of magnitude for many studied linear algebra benchmarks.

Our proposed framework makes a variety of applications practical in FHE. As a case in point, Figure 1 showcases the high-level input code and performance of both a dot product of two large vectors as well as a fully-connected layer in machine learning applications. The user of our system simply needs to describe their computation as a C++ program; no knowledge of complex FHE libraries is required, except for the desired level of security. For C++ code outlining a dot product of two encrypted vectors of length 500, our framework automatically generates a highly efficient circuit consisting of 922308 gates with 128 levels resulting in approximately one billion combined NTT and inverse NTT invocations.

2 Preliminaries

This section discusses different variants of homomorphic encryption and provides the motivation for adopting fully homomorphic encryption for general-purpose computation. Additionally, it provides theoretical details regarding the CGGI cryptosystem employed in this work.

Homomorphic Encryption

All encryption schemes that exhibit homomorphic properties enable meaningful computation directly on ciphertext data without revealing the underlying plaintext. The two variants of homomorphic encryption that support functionally complete sets of operations include leveled homomorphic encryption (LHE) and FHE. In both cases, ciphertexts are encoded as tuples of high-degree polynomials and adding or multiplying ciphertexts takes the form of polynomial addition or multiplication. These polynomials typically range from degree 2^{10} to 2^{17} and the coefficients are integers modulo q, which is a product of primes and typically hundreds of bits in length. In the encrypted domain, addition increases the ciphertext noise slightly, while multiplication is significantly more noisy. An unfortunate consequence of this ciphertext noise (which is necessary for security) is that the noise magnitude increases during each homomorphic arithmetic operation, and eventually the noise will corrupt the underlying plaintext message and prevent successful decryption with high probability. LHE can mitigate noise for a finite number of operations using a *modulus switching* technique, with larger encryption parameters allowing higher noise tolerance. However, larger parameters entail slower computation and higher memory consumption, limiting scalability for very deep circuits.

FHE solves the scalability issues inherent to LHE and allows for unbounded, arbitrary computation on encrypted data. First realized by Gentry in 2009 [30], *bootstrapping* is a noise mitigation technique that can be applied an infinite number of times, unlike modulus switching. In fact, any LHE scheme can be converted to an FHE scheme with the inclusion of bootstrapping. Nevertheless, the bootstrapping procedure itself is costly in terms of latency and remains a key bottleneck of all FHE constructions. Depending on the cryptosystem and chosen parameters, bootstrapping can take anywhere from several milliseconds [18] to minutes [31]. Therefore, the only way to achieve feasible FHE for general-purpose computation is to accelerate and optimize the bootstrapping mechanism.

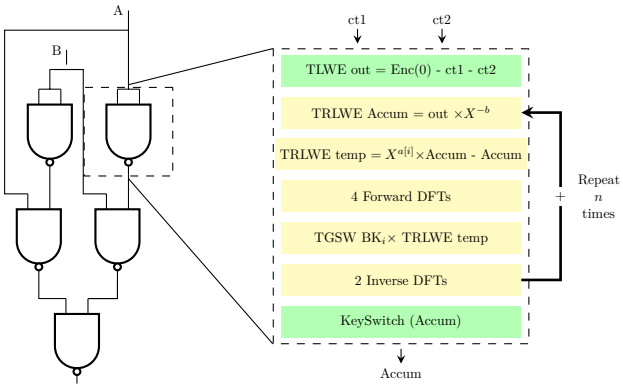


Figure 2: **Encrypted Logic Gate Evaluation**: All standard two-input logic gates begin with a series of linear operations between the input TLWE ciphertexts, followed by a bootstrapping procedure (executed by the looped instructions), and lastly a keyswitching operation. The steps in yellow represent operations associated with bootstrapping.

The CGGI Cryptosystem

Both the DM [27] and CGGI cryptosystems [18] possess bootstrapping routines that can be evaluated in up to tens of milliseconds on a CPU using modern open-source implementations such as Concrete [19] and OpenFHE [3], which is faster than other FHE cryptosystems. Also, while other schemes encrypt vectors of integers and floating point numbers, CGGI and DM are typically used to encrypt individual bits into a single ciphertext. Due to this encoding, the core encrypted operations take the form of Boolean gates, which are more flexible in terms of general computation than arithmetic operations over integers (e.g., encrypted comparisons are easily implemented using encrypted bits).

As discussed, to support unlimited computation depths, the FHE scheme must periodically invoke a bootstrapping operation to reset the amount of noise in the ciphertext. In the case of CGGI, which evaluates Boolean gates, bootstrapping must be performed every gate. As a result, evaluating a single homomorphic gate requires on the order of 2,000 polynomial multiplications [18], which are typically accomplished using the Discrete Fourier Transform (DFT). While this is an efficient algorithm for a single polynomial multiplication, even a small application could require billions of DFTs. For example, the computation graph for a single inner product of two vectors comprising 16 encrypted 16-bit numbers contains nearly 25,000 encrypted logic gates. Evaluating this circuit results in over 75 million invocations of the DFT. DM [27] was the first cryptosystem to introduce a *functional bootstrap* that refreshes noise while simultaneously evaluating a non-linear operation on the encrypted bits. In fact, this bootstrap is a necessary component of the computation for logic gates such as `NAND`.

CGGI improves upon this construction and generalizes it for all logic gates, including an encrypted `MUX` that is capable of obliviously choosing between two encrypted bits dependent on the underlying value of an encrypted selector bit. For all gates except the inverter, which is noiseless and linear, the bootstrapping operation comprises the majority of the gate's latency [41]. In turn, the core bottleneck of bootstrapping is the numerous polynomial multiplications between encrypted secret key components and input ciphertexts. Most FHE libraries perform these high-degree polynomial multiplications as element-wise multiplications in the DFT domain, which is asymptotically faster than textbook polynomial multiplication [14] [24]. Both the number theoretic transform (NTT) and fast fourier transform (FFT) can facilitate the forward and inverse domain conversions

for these purposes. However, the NTT is typically preferred over the FFT as it operates directly over integers. Moreover, FFT requires additional type conversions between integers and floating point numbers as FHE ciphertexts contain integer coefficients. As a result, FFT introduces small computation errors due to its reliance on floating point arithmetic.

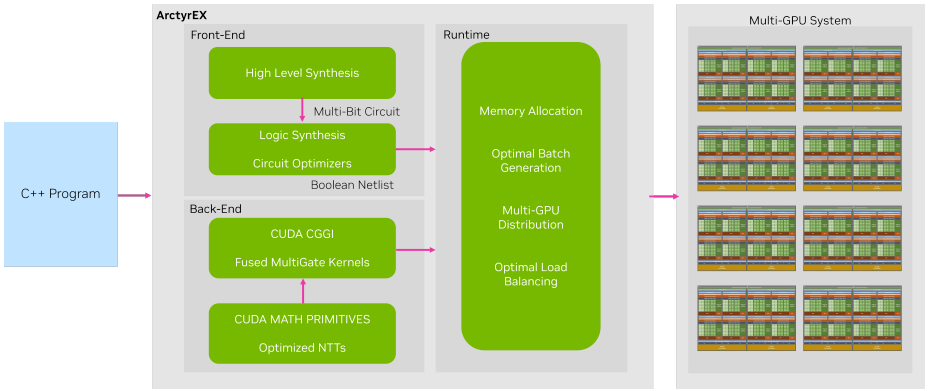
The CGGI cryptosystem employs different types of ciphertexts, each with different characteristics. The first type, known as TLWE ciphertexts, serve as the inputs and outputs of each homomorphic gate evaluation from a user perspective. TLWE ciphertexts are the smallest type that CGGI uses; at 128 bits of security, they consist of a single 630-degree polynomial with 32-bit coefficients and an extra 32-bit scalar term. However, these ciphertexts can not be used for nonlinear encrypted operations and are incapable of being used to evaluate a standard encrypted logic gate function (with the trivial `NOT` gate being the sole exception). Instead, these ciphertexts need to be transformed to TRLWE ciphertexts (i.e., Ring-LWE over the torus) that are larger in size. Typically, TRLWE ciphertexts are composed of a tuple of 1024-degree polynomials with 32-bit coefficients. The third type is TGSW ciphertexts, which are the largest and can conceptually be viewed as an array of TRLWE ciphertexts. The bootstrapping key, which is an encryption of the secret key, is composed of this type of ciphertexts. Importantly, TGSW ciphertexts can be multiplied directly with TRLWE ciphertexts, which is a necessary step of all bootstrapped gate evaluations. Figure 2 gives a high-level overview of the operations involved in a homomorphic `NAND` gate. All bootstrapped gates are evaluated in a similar way and only differ in the preliminary linear operations (i.e., the top green box in the figure).

Notably, certain classes of encrypted operations used for general purpose computation are well-suited for CGGI with binary ciphertexts, but are non-trivial using other cryptosystems that adopt multi-bit encodings. For instance, comparison operations, bitwise manipulations like shifting, and nonlinear functions such as the ReLU activation function in machine learning applications, can be computed directly without the need of costly polynomial approximations [11] [40]. Additionally, the requirement of executing hundreds of DFT transforms per bootstrap is particularly well-suited to GPUs due to the parallel nature of FFT and NTT. For these reasons, the CGGI cryptosystem is a good candidate for achieving accelerated general purpose computation on GPUs. CGGI can also support multi-bit encodings and employ a special programmable bootstrapping mechanism that evaluates univariate functions. However, only low precision is achievable with realistic parameters [21] and therefore this approach is better suited for specific applications rather than for arbitrary computation. Using a radix or CRT decomposition approach, which breaks up larger bit-width messages into chunks, univariate functions with larger precision inputs can be evaluated. However, we note that this approach scales poorly with increasing word size and generating larger LUTs is incredibly expensive [38]. As such, we strictly consider CGGI in gate bootstrapping mode with binary ciphertexts in this work specifically for this reason but note that the methodology proposed is readily extensible to support this scenario.

Threat Model

ArctyrEX generates code and provides an execution environment for a third-party cloud server to perform computations on encrypted data. We assume an honest-but-curious computing party, where the server can be trusted to do the expected computation but is incentivized to eavesdrop the sensitive user inputs. The server is aware of the underlying size and type of the data being manipulated (for example, integer, string, or class), as well as the evaluated algorithm. If the length of the data needs to be protected for a given application, we assume this is enforced on the client-side by introducing fixed input lengths. Our existing backend is based on the CGGI scheme [18] which reduces its security on the (R)LWE hard lattice problems.

Further, we assume that the cryptographic keys are generated by the client and the



(a) System Diagram

(b) Process Diagram

Figure 3: **System Overview:** Our proposed system is composed of three distinct layers that work together to realize an end-to-end framework for scalable encrypted computation. The frontend converts high-level programs to a logic circuit tuned for FHE. In turn, this logic circuit is parsed by the middle layer, which executes a coordination algorithm that partitions each level of the circuit into shares and assigns them to multiple GPUs. The back-end enables outsourcing computationally expensive FHE operations in each share to the GPUs.

secret key is not disclosed to another party. The evaluation key material, composed of the bootstrapping key and keyswitching key, required for encrypted logic gate evaluation, is uploaded to the cloud server. Notably, these keys are solely for facilitating encrypted operations and do not allow the cloud server to decrypt any input, intermediate, or output data.

Lastly, all benchmarks used to evaluate ArctyrEX assume that the client owns the input data, which is sensitive and therefore encrypted before uploading to the cloud server. For the neural network inference program, we analyze two variants that differ depending on which party owns the network weights. In one scenario, the client provides both the weights and the data to be classified as encrypted inputs. In the other case, the cloud owns the model weights and the client only provides the data to be classified. Additionally, ArctyrEX does not exhibit functional privacy and the cloud server has knowledge of the algorithm being executed.

3 System Design for Accelerated Encrypted Execution

ArctyrEX is an end-to-end framework that allows users to seamlessly convert high-level programs written in C++ to a sequence of GPU-friendly FHE Boolean operations leveraging the CGGI cryptosystem. An overview of the system and a process diagram illustrating the flow of data are depicted in Figure 3, illustrating the capabilities of the frontend, runtime schedule coordination, and backend operations as well as how these subsystems communicate and work together. The figure also shows the process of how a user's input program is converted to a multi-bit representation (i.e., Verilog code that operates over bit arrays) and then to an encrypted Boolean circuit and finally executed by a multi-GPU system. Our proposed frontend tackles challenges associated with leveraging

CGGI from a user perspective, such as adapting to the Boolean circuit model. In this section, we identify desirable circuit characteristics for efficient execution on GPUs and describe key aspects of synthesis to convert input programs to FHE code for outsourced computation.

Optimal FHE Circuit Characteristics

```
void full_gemm(short x[100], short y[100], short res[100]) {
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
      res[10*i + j] = 0;
      for (int k = 0; k < 10; k++) {
        res[10*i + j] = res[10*i + j] + x[i*10 + k] * y[k*10 + j];
      } } } }
```

```
int lr_inference(int data[4], int weights[4], int bias) {
  int product = 0;
  for (int i = 0; i < 4; i++)
    product = product + data[i] * weights[i];
  product = product + bias;
  // Sigmoid: scaled Maclaurin Series to stay in integer domain
  // s(x) = 40320 + 20160*x - 1680*x^3 + 168*x^5 - 17*x^7
  int temp = 40320;
  int squared = product * product;
  temp += 20160 * product;
  temp -= squared * product * 1680;
  squared *= squared;
  temp += squared * product * 168;
  squared *= product * product;
  product = temp - squared * 17;
  // Client post-processes score by dividing by 80640
  return product;
}
```

Figure 4: C++ functions for General Matrix to Matrix Multiplication (GEMM), or matrix multiplication, and Logistic Regression (LR) Inference.

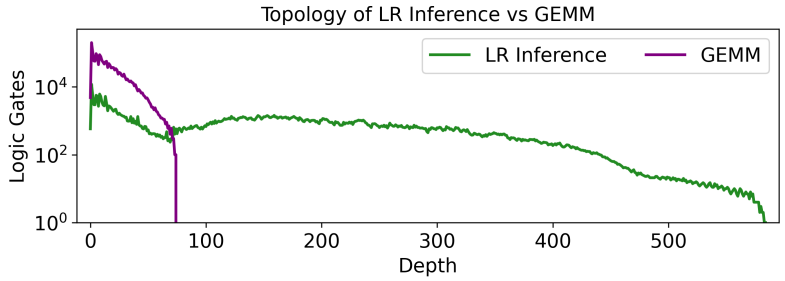


Figure 5: **Circuit-level parallelism**: Visualizing the circuit topology (i.e., the width of the circuit at all points) of GEMM versus LR inference shows that GEMM is ideal for parallel evaluation, while LR is less suitable.

One of the challenges for achieving efficient encrypted computation with the CGGI cryptosystem involves exploiting *circuit-level parallelism* at the logic gate level. Essentially, any number of gates with resolved dependencies (e.g., all input wires have been loaded with encrypted ciphertexts) can be executed in parallel as they are entirely independent. For CPU-based systems with a limited number of cores, this parallelism is sufficient to effectively saturate the available CPUs without any significant optimizations at the logic synthesis or application level. However, high-performance computing systems that leverage hundreds

of CPU cores or incorporate GPUs require much higher degrees of circuit-level parallelism to achieve high efficiency. For these systems, the characteristics of the underlying Boolean circuit become much more important, therefore avoiding sub-optimal configurations is a critical concern. For example, using the functions in Figure 4, we present the width of each circuit level for a 10×10 matrix multiplication as well as a logistic regression (LR) inference in Figure 5. The matrix multiplication benchmark represents ideal circuit characteristics for parallel execution as the majority of levels are very wide (the largest being nearly 200,000 gates in width), and the critical path is relatively short. On the other hand, LR inference has over 500 levels (resulting in a much longer critical path) and the width of each level is considerably shorter than those in the matrix multiplication circuit. Another type of circuit configuration ill-suited for systems that can exploit high degrees of parallelism is circuits that adopt cascading. Cascaded circuits consist of duplicated subcircuits chained together in a sequential way. These circuits typically have a long critical path and each level of the circuit is narrow, limiting the number of gates that can be evaluated in parallel at any given time.

Likewise, not all encrypted gates have the same execution time. For instance, `NOT` gates are significantly faster than other gates because they do not require any bootstrapping, while `MUX` gates are approximately twice as expensive as standard gates (like `AND` and `OR` gates). Efficient FHE circuit generation should take into account these differences in gate efficiency.

Strategies for Developing HLS-Compliant Programs

In terms of input programs, a strict requirement is that it must be synthesizable in order to generate a Boolean circuit that can be evaluated with FHE. Essentially, no dynamic memory constructs can be used as all data sizes need to be known at compile time to generate a fixed hardware circuit. Another constraint imposed by XLS is that standard C++ standard library (STL) functions and constructs can not be employed. As a case in point, the `sqrt` function defined by the `math.h` STL header can not be used directly. Instead, the square root operation can be implemented as a series of shifts and subtractions [35].

```
1  int partial_mm(int x[10], int y[10]) {
2    int res = 0;
3    for (int i = 0; i < 10; i++) {
4      res = res + x[i] * y[i];
5    }
6    return res;
7  }
```

Figure 6: Partial Matrix Multiplication HLS Kernel

Additionally, the way that an algorithm is expressed can have an impact on the HLS procedures that map the program to a Boolean netlist. For complex algorithms that contain several thousand loop iterations, the synthesis toolchain itself may be unable to successfully generate a circuit due to strict constraints on loop unrolling. In this case, we note that manually unrolling a number of iterations can solve the problem to bypass such constraints. However, for very large loops, this technique is quite cumbersome for developers as the loop body becomes large and redundant, while the time required for HLS and logic synthesis increases. We observe that we can overcome this challenge by splitting an algorithm into multiple HLS functions and invoking them one after the other in the encrypted application. Figure 6 demonstrates this strategy; the kernel implements the inner loop of a 10×10 matrix multiplication and can be invoked multiple times to evaluate the full procedure. The downside of this approach is that an additional setup cost is

incurred regarding circuit preprocessing and memory allocation, but we observe that a vast majority of the overall evaluation time is spent evaluating the circuit homomorphically.

Synthesizing FHE-friendly Circuits

The conversion process from a C++ program to an equivalent FHE algorithm can be completed in two distinct steps borrowed from modern hardware design paradigms: high-level synthesis (HLS) followed by logic or register transfer level (RTL) synthesis. While any HLS tool can be used for this purpose, we employ the Google XLS framework [34], which is a fast and efficient open-source HLS engine that can be used to rapidly generate synthesizable Verilog code. This Verilog code serves as an intermediate representation and describes the circuit functionality, which is then transformed by a logic synthesis tool to generate the actual Boolean netlist.

We utilize the open-source Yosys Open Synthesis Suite to facilitate this process and perform crucial circuit-level optimizations [63]. However, all existing logic synthesis tools, including Yosys, are tailored specifically for physical hardware development and optimize for several constraints that are not relevant to virtual FHE circuits (such as minimizing area or reducing clock cycle latency). The most relevant factors for optimal FHE circuit generation are minimizing the critical path delay, which is luckily a goal shared with actual hardware development, and prioritizing gates that run efficiently in the encrypted domain. Similarly to techniques introduced by the Google Transpiler [35], we can configure the logic synthesis tool to choose FHE-friendly gates by encoding the relative costs of each gate type as a function of area. For instance, we assign the multiplexer gate to be twice as big as the standard two-input gates to reflect the fact that the latency of the `MUX` is twice as slow as a standard gate.

Where prior work has adopted generic synthesis scripts for generating netlists for FHE evaluation [35, 37], our synthesis flow: (1) reduces the time required to generate the netlist relative to the Yosys generic synthesis script, and (2) results in more efficient circuits for FHE. The core optimizations that we utilize with Yosys include functional and word-size reduction, removing redundant logic, and omitting unreachable branches in decision trees. Compared to the baseline Google XLS logic optimizations, we observe a reduction of about 40% in the overall size of the circuit for a dot product of two vectors with length 500. However, we note that the Google XLS logic optimizer is more lightweight and can process the encrypted circuit about twice as fast. We emphasize that this process is a one-time cost; after the circuit is processed, it can be executed using an arbitrary number of inputs.

4 Novel Scheduling Algorithm for Scalable Evaluation

The ArctyrEX runtime library implements our proposed scheduler that allows homomorphic applications to utilize multiple computing resources with high scalability.

Strategies for Evaluating FHE Circuits

After the Boolean netlist has been generated by the frontend compiler, and before encrypted computation can be carried out, we need to translate each gate to the encrypted domain. This process involves traversing the circuit, which is represented as a directed-acyclic graph (DAG), and mapping each node to the equivalent CGGI gate function. All wires become ciphertext data, the inputs are loaded with encryptions provided by the client, while the outputs are communicated back to the client for decryption after circuit evaluation.

The intuitive approach for providing the computing party with an executable FHE circuit is to simply generate code that invokes the encrypted gate functions using the underlying backend directly one after the other. This approach works well for small programs where performance is not critical, but is ill-suited for non-trivial programs. For

complex programs, the generated FHE code can grow to millions of lines in length, as each logic gate in the circuit would require 2-3 lines of code on average.

Our key observation is that it is more efficient to avoid code generation entirely and incorporate a scheduler that traverses the DAG and distributes each gate to additional workers that exclusively run the corresponding FHE logic gate function. In this approach, gates that are ready to be evaluated can be distributed across a set of workers to exploit the circuit-level parallelism inherent in all applications. We remark that a similar methodology is employed by the Google FHE transpiler [35] and is referred to as *interpreter mode*. However, their strategy of distributing gates one at a time is not feasible when the workers constitute GPUs. Previous GPU-centric CGGI implementations as well as our proposed implementation (described in Section 5) can execute one homomorphic logic gate per streaming multiprocessor (SM) concurrently. In the case of an NVIDIA A100 GPU, 108 homomorphic logic gates are the least number required to achieve 100% device utilization at any given time. Thus, only one SM could be engaged if gates are assigned one at a time, resulting in extremely inefficient evaluation. Further, interpreter mode creates new ciphertext objects and allocates more memory as needed throughout circuit evaluation. While this technique is suitable for CPU workers, it therefore creates a prohibitive bottleneck on GPUs as memory allocation and ciphertext transfers between the host and the device are costly.

Algorithm 1: Runtime Library Circuit Preprocessing

```
   Input: circuit, numWorkers
   Output: levels
 1 levels = new Queue[circuit.depth()]
 2 readyQueue = new Queue
 3 pending = new HashMap < Gate, int >
 4 step = 1
 5 maxGates = 0
 6 for gate in circuit do
 7 |   pending[gate] = gate.numOperands
 8 |   if gate.numOperands == 0 then
 9 |   |   readyQueue.append(gate)
10 |   |   levels[0].append(gate)
11 while readyQueue is not empty do
12 |   if readyQueue.size() > maxGates then
13 |   |   maxGates = readyQueue.size()
14 |   nextReadyQueue = new Queue
15 |   while readyQueue is not empty do
16 |   |   Gate n = readyQueue.front()
17 |   |   readyQueue.pop() for gate in n.operands do
18 |   |   |   pending[gate] = pending[gate] − 1
19 |   |   |   if pending[gate] == 0 then
20 |   |   |   |   nextReadyQueue.append(gate)
21 |   while nextReadyQueue is not empty do
22 |   |   Gate n = nextReadyQueue.front()
23 |   |   nextReadyQueue.pop()
24 |   |   readyQueue.append(n)
25 |   |   levels[step].append(n)
26 |   step = step + 1
27 maxMemory = maxGates ∗ 2 ∗ sizeof(Ctxt)
28 Allocate maxMemory bytes
29 return
```

ArctyrEX Runtime Library

We propose a novel methodology for efficient evaluation of encrypted circuits on both CPU and GPU devices. The host thread parses the intermediate representation (IR) generated by the frontend, and generates a set of nodes stored using XLS data structures [34]. Each node contains an opcode, which defines the operation performed by the circuit gate, and its input operands, which are pointers to other XLS nodes.

The IR thus defines a sequence of gates which can be processed sequentially to generate a valid execution of the circuit. In order to introduce parallelism, we transform this ordered set of XLS nodes into a vector of circuit gates. In addition to logic gates, we also create gates which compute encrypted constant values and augment the frontend IR with gates which copy an input ciphertext into another one. These copies are used to integrate the retrieval of encrypted results as part of the circuit itself instead of having to extract individual ciphertexts after waiting for circuit termination.

To execute the circuit in parallel, we first dispatch all gates into *waves*. Each wave must be processed in-order, but all entries of a *wave* can be processed concurrently. Algorithm 1 shows the topological sort algorithm that we use to build the list of *waves*.

For each entry of the vector of gates, we compute its successors (gates depending on it), and count its predecessors (gates on which it depends). To assign gates to different waves, we create a FIFO of ready gates, which are gates with no remaining dependencies. We start by adding all gates with no input dependencies into this FIFO. Until the FIFO is empty, we remove the first entry n from the FIFO, and do the following:

- Assign n to the first wave if there are no input dependencies, or compute the max index of the wave of all predecessors, and add n to the next wave. All predecessors have an index or n would not be in the ready FIFO.
- We decrement the predecessor count of all successors of n. Any of these successors reaching a null predecessor count are put in the ready FIFO.

This algorithm terminates even if the circuit is not connected. As the IR can be processed sequentially one node after the other, there cannot be cycles in the circuit and all gates will be given an index. Since nodes are assigned to waves with indexes that are strictly greater than their predecessors, all entries in a wave are independent and can be processed concurrently, as long as the waves are processed in order. We thus automatically derived a parallel execution from the IR, based on the fact that the IR was a valid sequential execution and used node operands to compute dependencies. This algorithm has a linear complexity because each node is taken exactly once from the FIFO, and we decrement counters as many times as there are wires in the circuit. Partitioning the circuit into such waves provides concurrency which can be exploited to efficiently use a single GPU device. For multiple processing units, we dispatch waves over the different devices. A simple solution to dispatch a wave with N gates over K devices that consists of splitting it into roughly N/K gates per device, is illustrated in Figure 7.

Let us consider a wave with 43875 gates, composed of 2125 `AND` gates, 25000 `OR` gates and 16750 `NOT` gates. On two devices we could have 1356, 10465 and 769 gates of type `AND`, `OR` and `NOT` for device 0; and have 769, 14535 and 6633 gates of these types on device 1. This represents a total of 21938 gates on device 0, and 21937 on device 1, but we measured that device 0 and 1 respectively need 2.21 ms and 2.87 ms to process their portion of the wave. This 30% load imbalance is explained by the fact that `AND` and `OR` gates take the same time to process (about 0.19 us per gate), while it also takes 0.19 us to process 1024 `NOT` gates, which are non-bootstrapped. Equally dividing the gates between devices is therefore not a satisfactory approach, and only results in a 1.7x speedup with two devices.

We could consider the disparities between the different types of gates to evenly divide the load between the different devices based on performance models, but it would require an

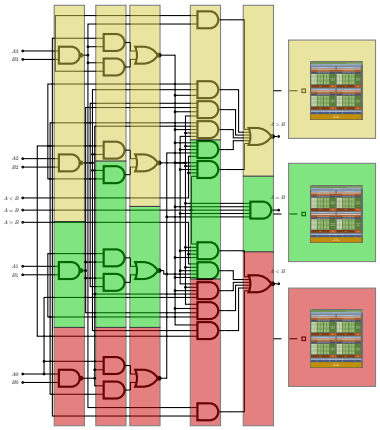


Figure 7: **Mapping gates to devices**: Gates are dispatched in independent waves, which are split across different processing units. In this example, we extract 5 waves which are spread over 3 GPUs that receive a similar workload.

extra training phase per gate type. This may be tedious and not reliable when combining multiple gates with different compute or memory bandwidth intensity. In practice, the number of gates is usually large enough that a simpler but effective solution is to assign the same number of identical gates on all processing units. In our previous example, this results in putting 1064 and 1063 `AND` gates respectively on devices 0 and 1, and putting 12500 `OR` gates and 8375 `NOT` gates on both devices 0 and 1. We then measure 2.54 ms of work on both devices, with a negligible difference of less than 0.2 us, which corresponds to a perfect balancing.

For each wave, ArctyrEX implements this strategy using a *hash-table* which associates a vector to each of the opcodes encountered in the wave. Each entry of the wave is then appended to the list which corresponds to its opcode. Considering that there are only 8 types of standard logic gates currently supported, and that this number would not grow significantly, appending an entry roughly has a constant complexity. This phase therefore also has an overall linear complexity. In Section 5 we will show that building such vectors of identical gates makes it straightforward to implement batched kernels which obtain much higher performance.

In this Section, we have shown how ArctyrEX converts the frontend IR into a well-balanced parallel workload. Provided CPUs with sufficient processing power, nothing prevents us from assigning them parts of the waves too. This could be done using performance models based on per-gate performance models, or more simply based on the respective peak performance of the different types of processing units. Our methodology is therefore suitable to address hybrid systems combining CPUs and GPUs.

5 A fully asynchronous cryptographic backend

In the previous Section, we described the circuit as a sequence of *waves* subdivided into smaller sets of homogeneous gates to obtain an efficient load balancing over the processing units. This Section details our native CUDA implementation of CGGI, and explains how we execute this workload as efficiently as possible thanks to a fully asynchronous implementation. We will now refer to these sets of concurrent homogeneous gates as *batched gates*.

Memory and Communication Considerations

Since we have covered how gates are batched for distribution to different processing units, we now describe how we can access data across the entire system. NVIDIA GPUs have a distinct memory hierarchy that differs in key ways from traditional CPUs. Inside a streaming multiprocessor (SM), there is a fast on-chip piece of memory partitioned between an L2 cache, and a resource called *shared memory*. These on-chip memories are much faster than global memory as they are part of the SM itself. In fact, shared memory latency is roughly 100x lower than un-cached global memory latency, provided efficient memory access patterns. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. In the case of the A100 GPU, the combined capacity of on-chip memory per SM is 192 kB. Global memory is the largest memory (40 or 80 GB for the A100) and resides off-chip, making it the slowest aside from accessing memory on the host [51].

Bootstrapping keys have a relatively large size, more than 100 MB for 128-bits of security. They cannot fit into GPU L2 caches, but are used for the majority of encrypted gate evaluations. Because of this, we replicate them in the global memory of all devices so that each can access the evaluation keys directly. We note that these keys are constant, and can be accessed concurrently within a device.

Ciphertexts are processed during circuit evaluation, and may be used simultaneously on different devices, or accessed from the host. We thus store them in pinned host memory, which is memory allocated with `cudaMallocHost()`). Ciphertexts are then cached into shared memory which is much faster and is located close to the GPU SMs. These kernels indeed have an extremely high arithmetic intensity and the PCI-e bandwidth consumption is limited, and the large amount of concurrency overlaps transfers with computation. This was verified experimentally by profiling a kernel that processes 1024 gates using the `ncu` tool. We observed that it only consumed 19.96 MB/s of "system memory" bandwidth, which is orders of magnitude lower than the available PCI-e bandwidth. Using pinned host memory to load the input ciphertexts is thus efficient enough, in spite of its simplicity.

A similar strategy is to use managed memory (also called *unified memory*, and allocated using `cudaMallocManaged`). Contrary to pinned memory where devices access host memory directly through the PCI-e bus, managed memory is kept coherent across the entire machine by the means of paging mechanisms. When a page fault occurs, the CUDA driver automatically fetches a valid copy of the page where the fault occurred. Subsequent accesses to the same page will occur at the speed of the memory embedded on the device, until the page is evicted from the device.

Both managed memory and pinned host memory incur a significant overhead per allocation, so that we do not allocate all ciphertexts individually, but group these thanks to pooled memory allocators. This pooling mechanism may introduce false sharing issues, but effectively amortizes allocation overhead, which remains noticeable with pinned host memory, but is several orders of magnitude lower than the time required to evaluate the circuit. Memory pages allocated with managed memory and modified concurrently by multiple devices may bounce from one device to another, and have a severe impact on performance.

In practice, we observe similar performance for an encrypted dot product over 8 GPUs with both strategies. With pinned memory, circuit evaluation takes 14.8 s, compared to 15.1 s with managed memory. Allocating 1 GB of pinned host memory however takes 0.4 s, but is negligible with managed memory. Due to the expected page faults when using managed memory on multiple devices, we observe some slightly imperfect parallelism, while it is flawless with pinned memory. ArctyrEX therefore allows user to store ciphertexts either in host pinned or managed memory, for example depending on the amount of system memory which can limit the availability of pinned memory. All experiments presented in the rest of this paper use pinned host memory.

Coordinating Multiple Devices

A strawman approach to assign tasks to multiple workers involves having a single producer thread and a set of worker/consumer threads. When using multiple GPUs, each worker thread consumes an assigned batch from the producer and outsources the computation to a dedicated GPU. This approach is simple to implement, but requires numerous synchronizations between CPU threads, which negatively impacts scalability by introducing idle periods on the GPUs when CPU threads fail to provide them computation.

Algorithm 2: ArctyrEX Dispatch Algorithm

```
   Input: levels, numWorkers
1  nsteps = batched_waves.size()
2  for wave in batched_waves do
3      for batch_id in wave.size() do
4          // Dispatch GPU batches across GPUs
5          dev_id = batch_id % numWorkers;
6          process_batch(batches[batch_id], dev_id);
7      need_sync = (step == nsteps − 1);
8      fence_on_devices(need_sync);
```

Conversely, a more intuitive method involves utilizing a single host thread that will submit work asynchronously to different devices. On each device, we create a pool of CUDA streams, so that we can submit multiple concurrent CUDA kernels on this device. The execution of a single wave therefore consists of taking each individual batched kernel from the wave, selecting a CUDA stream on the device on which our scheduling algorithm assigned the batched kernel (*e.g.,* with a round-robin strategy), and submit the kernel in this CUDA stream. Since waves must be executed in-order, we need to ensure that the execution of a wave does not start until the previous wave has been fully processed. A simple approach would consist of submitting all kernels in a wave on all devices, and then having the host thread wait for the completion of all work on all devices. Waiting for computation to complete from the host however introduces some inefficiency, as devices become idle during the synchronization phase, until the next phase has been submitted. Any potential load imbalance or jitter on a device may also reflect on other devices which could wait longer than expected to get more work. Instead of blocking devices, we have therefore implemented a non-blocking synchronization fence primitive which ensures that the work in a CUDA stream cannot start running until all work submitted previously in all other streams has been done. These fences are implemented by the means of CUDA events which are asynchronously inserted in the CUDA streams. After inserting an event in each stream, we insert a non blocking CUDA operation which synchronizes one of our CUDA streams with all of these events. We then insert another event in that stream, and make sure all other streams wait for that event. Event insertions and dependency declarations between an event and a stream can be performed asynchronously, ahead of time, and therefore do not require the host thread to block during the execution. These event-based synchronizations are implemented using hardware features, which is much more efficient that having the host thread block the entire device. This ensures that successive waves can be executed in order, without ever blocking the submission flow of asynchronous operations, until the very end of the circuit evaluation.

With this distribution methodology (Algorithm 2), we observe a speedup of approximately 12% over the strawman approach for an encrypted dot product benchmark executed on an NVIDIA DGX A100. This may appear to be a moderate improvement, but more than 99% of the circuit evaluation is spent executing CUDA kernels. Therefore, we have a close to optimal scheduling strategy over multiple devices, which is essential for the

scalability of ArctyrEX according to Amdhal's law. This also indicates that the latency of result retrieval and synchronizations are almost completely hidden.

Batched Kernels

Due to the relatively small size of TFHE ciphertexts (compared to other FHE schemes), it is possible to process many FHE gate operations at the same time on GPUs over a large number of ciphertexts. Prior works have either launched a separate kernel for every gate evaluation [26] or allow for "vectorized" gates (such as performing a bitwise NAND between two 32-bit ciphertext arrays) [50]. Conversely, we observe that a better approach for general computation is to leverage a kernel capable of executing arbitrary numbers of gates of any supported type. The ArctyrEX backend utilizes a single kernel for each batch of gates that launches with N thread-blocks of 512 threads each, where N indicates the number of gates. This approach is more performant compared to the cuFHE library that initiates host-to-device and device-to-host transfers for each logic gate. This allows each worker in the runtime environment to launch one kernel for each batch received from the coordinator, avoiding additional kernel launch overheads. Additionally, this technique also allows the GPU to determine the best utilization strategy for the SMs, instead of relying on users to distribute gates on a per SM basis. Grouping gates into homogeneous gates saves memory bandwidth as we only copy the opcode value once per batched kernel, and the generated code is more regular and requires less registers, increasing the *occupancy* of our CUDA kernels [52].

Designing batched CUDA kernels which do not require blocking the submitting host thread is also challenging. These kernels indeed need to access buffers with the description of the work to perform, such as the location of the input ciphertexts. We therefore adopt a strategy which consists of assigning such a buffer to each CUDA stream of our pool, and fills them asynchronously from the host using a *host callback*. As a result, our asynchronous batched kernels consist of 1) selecting a CUDA stream on the target device, 2) submitting a host callback that will update the buffer associated to this stream, and 3) launching a CUDA kernel in this stream which will process the batch described using this buffer. Assigning each CUDA stream a unique buffer requires a limited memory footprint, and ensures there is no concurrent buffer update. This also avoids the use of relatively expensive asynchronous allocations around all asynchronous kernels.

NTT Implementation Details

The performance of CGGI bootstrapping is largely determined by the DFT used to perform polynomial multiplication. Both nuFHE [50] and cuFHE [26] use the NTT for this operation, and employ the same strategy in terms of NTT parameters. We opt to use these parameters as well, since they provide multiple key optimizations that reduce the NTT latency. First, we utilize the modulus $Q = 2^{64} - 2^{32} + 1$, which simplifies modular reduction and supports NTTs up to size 2^{32}. Lastly, we use the primitive element $g = 12037493425763644479$ that allows most multiplications in the NTT algorithm to become bitshifts modulo Q.

6 Experimental Evaluation

We employ a series of benchmarks representing realistic computational workloads with FHE to demonstrate the efficacy of ArctyrEX, encompassing areas such as privacy-preserving machine learning, linear algebra applications, and cryptographic benchmarks. All experiments were run on an NVIDIA DGX A100, which consists of 8 A100 GPUs and a dual-socket AMD EPYC 7742 CPU with 64 cores each (a total of 128-cores running 256

threads with simultaneous multithreading). We believe that this high-performance CPU comprises a fair baseline to our CUDA-accelerated framework running on up to 8 A100 GPUs as it constitutes a commonly used datacenter CPU for high-performance applications. Additionally, we compare each benchmark against the state-of-the-art Concrete library, which also incorporates GPU acceleration capabilities for the CGGI cryptosystem. When comparing directly against Concrete, we utilize 2 A100 GPUs for both the ArctyrEX and Concrete backends. Unless otherwise indicated, all benchmarks were run with parameters corresponding to 128 bits of security based on the BKZ-beta classical cost model provided by the state-of-the-art LWE estimator framework [4]. Specifically, for TRLWE ciphertexts used in bootstrapping, we utilize a ring dimension of 1024 and for TLWE ciphertexts we utilize a dimension of 630. These are the same TRLWE/TLWE parameters employed by the TFHE library [18] for their parameter set corresponding to 128 bits of security.

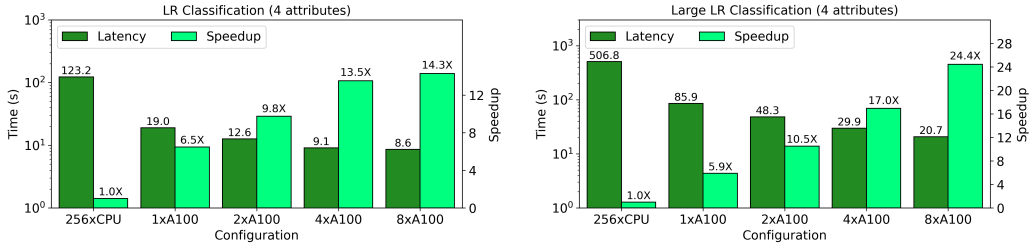


Figure 8: **Logistic Regression Inference**: We employ 32-bit words for the small approximation (i.e., evaluating four terms of the Maclaurin series) and a 64-bit wordsize for the large approximation that evaluates six terms of the Maclaurin series to avoid overflow. We observe a better scaling trend for the higher accuracy LR because it exhibits wider levels.

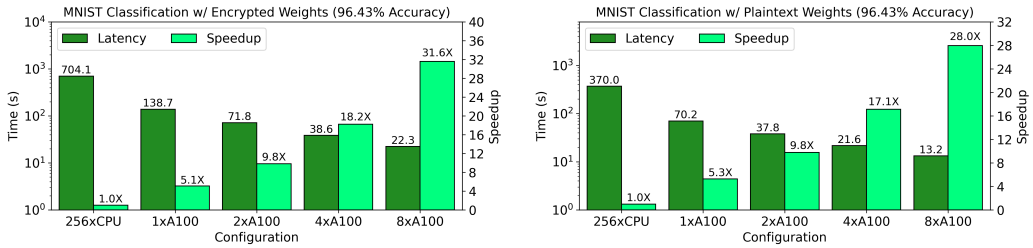


Figure 9: **Neural Network Inference**: The encrypted weight variant represents the scenario where the computing party does not own the model, unlike the variant with plaintext weights. We observe a roughly 2× speedup when plaintext weights are used.

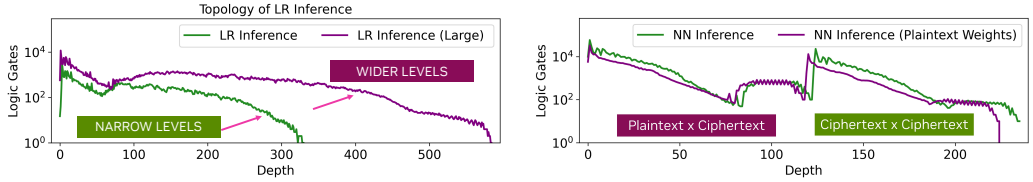


Figure 10: **Topology of Machine Learning Benchmarks**: For LR inference, the large variant uses a more accurate sigmoid approximation. It is much deeper due to a larger word size and more polynomial terms evaluated. The neural network plaintext weight variant exhibits a shorter critical path and is composed of much fewer gates overall.

Encrypted Linear Algebra Benchmarks

Motivated by image processing and machine learning, we benchmark ArctyrEX using three tensor multiplication algorithms on 16-bit encrypted data: a dot product of two vectors (length 500), a matrix-vector multiplication (125-size vector and 125×4 matrix), and a matrix multiplication (10×10 matrices). Our detailed analysis is presented in Appendix A.

In summary, ArctyrEX on one A100 GPU runs 6.1× faster than on a 256-thread CPU, and 8 A100s run 42.5× faster, following a linear speedup with more GPUs. Vector addition is 4× faster on 8 GPUs. Concrete can perform vector addition about 35.5× faster than ArctyrEX, due to its ability to pack multi-bit values in a single ciphertext and its natively support for addition. Conversely, ArctyrEX is 29.3× faster than Concrete for vector dot product. For matrix-vector product, ArctyrEX is 11.6× faster than Concrete and 14× faster for 10×10 matrix multiplication.

Encrypted Machine Learning Applications

One of the most widely explored use-cases for FHE is privacy-preserving machine learning as a service. We consider two distinct scenarios: one where the model is owned by the cloud server and another where the client or a third party owns the ML model. For the former, the network parameters are kept in plaintext form and can benefit from faster plaintext-ciphertext operations. Conversely, when the cloud doesn't own the model, the network parameters are encrypted as well as the classification inputs. The majority of existing works demonstrating ML inference with FHE adopt the first scenario as it is generally much more efficient [29, 12, 23].

Our analysis considers two important machine learning procedures for encrypted classification in the form of logistic regression (LR) inference and neural network (NN) inference. Logistic regression is a popular algorithm geared for binary classification and consists of a fully-connected layer (a matrix-vector product between trained weights and input data) as well as a non-linear sigmoid activation function. For this benchmark, we perform binary classification for datasets with four attributes, such as the Iris dataset [28]. The key bottleneck in encrypted LR inference is approximating the sigmoid function ($\frac{1}{1+e^{-x}}$), since it is not possible to evaluate it directly. Therefore, we employ a polynomial approximation by evaluating terms of the Maclaurin series. In general, when approximating nonlinear activation functions, there is a trade-off between accuracy and computational complexity. We show this trade-off with ArctyrEX through the use of an approximation that evaluates the first four terms, and one that evaluates the first six. Figure 8 shows diminishing returns when increasing the number of GPUs due to the large critical path and relatively thin circuit levels of the benchmark in Figure 10. Using 8 GPUs with the more accurate sigmoid approximation still outperforms the CPU implementation with 256 threads by 24.4×. Concrete is approximately 4× faster than ArctyrEX for the less accurate approximation, owing to its high efficiency for multiplications with constants and additions. However, Concrete is unable to support the larger bit-width required for the more accurate approximation. On the other hand, ArctyrEX supports arbitrary precision and bit-widths.

Neural networks constitute a more complex benchmark than logistic regression, which can be viewed as a small single layer neural network. These ML architectures consist of a series of linear (i.e., fully-connected and convolutional layers) and non-linear layers (i.e., activation functions). We employ the same network architecture used by FHE-DiNN [12] for our evaluations. The network consists of two fully-connected layers with a sign activation function to classify the MNIST dataset of handwritten digits. While fully-connected layers are well-suited to HE cryptosystems that natively support addition and multiplication primitives, the sign activation function is very well-suited for CGGI, as the

Table 1: AES-128 Decryption: ArctyrEX performs competitively with state-of-the-art application-specific frameworks for transciphering with AES. We do not consider key-scheduling as prior works have opted to omit this step in favor of sending encryptions of the pre-generated round keys. All frameworks use parameter sets corresponding to 128 bits of security, with the exception of Gentry et al. [32] which is configured for approximately 123 bits of security.

Framework	Scheme	Hand-Tuned	Time (s)
ArctyrEX (1xA100)	CGGI	No	45
ArctyrEX (2xA100)	**CGGI**	**No**	**30**
Google Transpiler [35]	CGGI	No	243
HELM [36]	CGGI	No	86
Concrete (2xA100) [19]	CGGI	No	9211
Stracovsky et al. [58]	CGGI	Yes	252
Trama et al. [59]	CGGI	Yes	28
Wang et al. [61]	CGGI	Yes	28
Fregata [62]	**CGGI**	**Yes**	**9**
Gentry et al. [32]	BGV	Yes	1050
Aharoni et al. (GCM) [2]	CKKS	Yes	417
Aharoni et al. (CTR) [2]	CKKS	Yes	99

encrypted sign bit can be extracted for free since each individual ciphertext encodes one bit of information. We consider two variants of this network that differ in whether or not the network parameters are encrypted and both achieve an accuracy of 96% for MNIST classification. The execution times across both configurations are depicted in Figure 9 while Figure 10 presents the characteristics of these workloads. As expected, the variant where the cloud server does not own the proprietary network (i.e., using encrypted weights) has approximately 2× higher latency because of the increased number of ciphertext-ciphertext operations. Relative to Concrete, we observe that ArctyrEX is two orders of magnitude slower for the variant with plaintext weights. This is primarily attributed to the low cost of constant multiplications with Concrete. On the other hand, ArctyrEX is approximately 17× faster for the variant with encrypted weights. We remark that both FHE-DiNN and REDsec [29] achieve lower overall latencies for high-accuracy MNIST classification with plaintext weights (1.7 seconds and 3.6 seconds respectively), but utilize techniques only applicable to neural network inference. For instance, both approaches utilize a modified bootstrapping technique to evaluate the sign activation function across multi-bit encodings. Other works aim to maximize inference throughput with other FHE schemes that allow for batching and SIMD-style computation. CryptoNets [33] and Faster CryptoNets [23] utilize BFV to classify MNIST in 39 and 250 seconds respectively. However, both can classify thousands of independent inputs simultaneously.

We emphasize that the inclusion of these two ML benchmarks aims to showcase the AI capabilities with CGGI. Other AI-based demonstrations using alternative FHE schemes, such as CKKS [43, 42, 44, 8, 68, 66, 53], are listed in Appendix C.

AES-128 Decryption

At first glance, it seems odd to compute an encryption algorithm homomorphically when the data is already encrypted. However, this enables an exciting strategy called *transciphering* that dramatically reduces the communication overhead associated with FHE [5, 22]. Instead of sending large HE ciphertexts to the cloud, the client can send encryptions generated with a traditional block or stream ciphers that result in little to no data expansion. Then, the

cloud can *homomorphically* decrypt the received symmetric ciphertext by evaluating the corresponding decryption algorithm of the chosen cipher using an homomorphic encryption of the symmetric key. For the CGGI cryptosystem at 128 bits of security, this strategy can decrease the communication overhead associated with the client sending encrypted inputs by a factor of 20000×.[1]

Our analysis employs the Advanced Encryption Standard (AES) cipher, which is a ubiquitous block cipher. AES is well-suited to evaluate CGGI cryptosystems because each round is primarily composed of bitwise operations. In terms of design considerations, we implement the SubBytes step as a sequence of lookups instead of computing the S-boxes on the fly; we note that computing the Galois Field arithmetic, and computing the multiplicative inverse, is more expensive than performing a lookup that maps an encrypted input byte to one of 256 precomputed entries. Notably, only the decryption circuit is needed to perform transciphering as this operation converts AES ciphertexts to HE ciphertexts.

Table 2: AES-128 Throughput: While AES decryption does not exhibit enough parallelism to saturate more than two A100 GPUs, we can use the remaining GPUs to decrypt more blocks in parallel. Our approach is beneficial for downstream applications that employ TFHE.

Framework	**Blocks**	**Scheme**	**Throughput (blocks/s)**
ArctyrEX (1xA100)	1	CGGI	0.02
ArctyrEX (2xA100)	1	CGGI	0.03
ArctyrEX (4xA100)	2	CGGI	0.07
ArctyrEX (8xA100)	4	CGGI	0.13
Gentry et al. [32]	180	BGV	0.17
Aharoni et al. (GCM) [2]	32768	CKKS	78.6
Aharoni et al. (CTR) [2]	32768	CKKS	331

Table 1 depicts the latency of ArctyrEX for decrypting one block homomorphically with AES-128 for one and two A100 GPUs. Additionally, Table 2 shows the throughput achieved with up to 8 A100s. We note that the decryption circuit is not wide enough to yield any additional benefits for more than two A100 GPUs. ArctyrEX exhibits significantly lower latency relative to the original implementation of AES with FHE [32], which utilizes the BGV cryptosystem (albeit on a Lenovo X230 laptop with an Intel i5 processor). Additionally, ArctyrEX outperforms several works based on CGGI [36, 35, 58, 19]. Both Trama et al. [59] and Wang et al. [61] are marginally faster, but both are application-specific and employ optimization techniques unique to AES evaluation. Fregata [62] is also application-specific and can evaluate the AES decryption circuit homomorphically in approximately 9 seconds on a multi-core CPU. Notably, Fregata employs a bespoke packing strategy which may not be amenable to downstream operations. On the other hand, the ArctyrEX AES evaluation yields 128 output ciphertexts per block, which is compatible with all downstream gates. Finally, Aharoni et al. [2] implement GPU-accelerated AES-GCM and AES-CTR decryption with the CKKS cryptosystem on an A100 GPU and achieve high throughput due to the ability to batch tens of thousands of blocks simultaneously. We note that the problem of transciphering with authenticated encryption (e.g., GCM mode), which constitutes a different decryption algorithm, is an interesting and orthogonal direction relative to our approach and those employed by the other works. Nevertheless, ArctyrEX achieves lower latency for a small number of blocks.

[1] For the default LWE dimension of 630 with 32-bit polynomial coefficients, each plaintext bit expands to $(630 + 1) \cdot 32$ bits when encrypted [18, 19].

Security and Latency Tradeoffs

All experiments presented previously utilized a security level of 128 bits; however, at the cost of lower security, one can decrease the polynomial degree and therefore the ciphertext size. This results in faster HE operations and reduces the overhead of bootstrapping. For a security level of 110 bits, we utilize an LWE dimension of 512 with a noise rate of 2^{-15}. Overall, we find that the 128-bit parameter set results in a latency increase of approximately 1.3× for all GPU configurations for LR inference.

7 Related Works

Prior works can be divided into two categories: FHE compilers for general-purpose computation and GPU frameworks that reduce the latency or improve throughput of homomorphic operations. The former category targets the usability issue inherent in FHE and explores automatic application-level optimizations to facilitate efficient execution for the target backend. The ArctyrEX frontend and middle-layer address these challenges as well, and can be directly compared prior works in this line of research. The latter category includes works that focus on FHE acceleration using both software and hardware techniques at the primitive level, and are also comparable to our proposed backend.

FHE Compilers

The Cingulata framework (formerly Armadillo [13]) allows users to map C++ code into a sequence of `AND` and `XOR` gates. Cingulata works with binary FHE contexts using the TFHE library (which implements CGGI) and a custom BFV implementation as its backends. Compared to ArctyrEX, Cingulata only supports single-core CPU execution for CGGI. The BFV mode is parallelized on CPUs, but does not support bootstrapping and hence cannot be used for arbitrary general-purpose computation.

E^3 is a C++ library that introduces encrypted data types for leveraging FHE in general applications [17]. It supports a variety of backends, including TFHE, Microsoft SEAL, and HElib, encompassing all major FHE schemes. Unlike ArctyrEX, E^3 uses a direct mapping to hardcoded FHE functional units and does not offer an optimizing compiler. It also does not support any GPU-friendly cryptographic backends and no parallelization is included.

Google's FHE Transpiler [35] and Romeo [37] leverage logic synthesis and optimizations to generate FHE programs for general computation. However, both works employ generic synthesis scripts that include optimizations not relevant to encrypted computation. The FHE Transpiler targets TFHE and the OpenFHE implementations of the CGGI cryptosystem as backends, and can evaluate multiple gates in parallel using interpreter mode. However, it does not support GPUs and its parallelization strategy is not suited for them, yielding very low device utilization. Likewise, Romeo targets TFHE and generates an FHE program instead of interpreting it. This approach, however, does not scale for large programs or HPC systems as described in Section 4. Conversely, ArctyrEX offers a novel dispatch strategy and multigate kernels that can efficiently compute batches of any set of gates.

FHE Acceleration Frameworks

The cuFHE [26] and nuFHE [50] libraries incorporate GPU acceleration of the CGGI cryptosystem. The former is a proof-of-concept library that implements logic gate evaluations on a single NVIDIA GPU. However, cuFHE is not configurable (i.e., only supports 80 bits of security), has non-optimal data transfers and requires frequent high-cost synchronization between GPU and CPU. Each cuFHE gate evaluation requires all ciphertext inputs be

Table 3: Comparisons with existing backends for 32-bit arithmetic operations (taken from Morshed et al. [46]). Most backends are configured for 80 bits of security, while ArctyrEX and Narisada et al. [49] configured for 128 bits of security. All backends run on a single GPU besides the CPU-based TFHE that runs on a single thread. For the GPU frameworks, a technology scaling factor was introduced for fair comparisons (defined as the number of SMs in our A100 GPU divided by the number of SMs of the target GPU).

Library	Add (s)	Mult (s)
ArctyrEX	1.70	2.83
Narisada et al. [49]	0.10	2.22
FHE Transpiler [35]	6.53	13.56
Morshed et al. [46]	1.47	25.13
TFHE [18]	7.04	489.93
nuFHE [50]	3.08	137.78
cuFHE [26]	1.50	97.5
REDcuFHE [29]	1.55	99.21

copied from the host to the device, and each output is copied back from the device to the host. This approach is impractical for realistic circuit evaluation, as it yields millions of large ciphertext transfers between the CPU and GPU. Lastly, not all cuFHE computations are outsourced to the GPU and the CPU needs to perform certain operations (such as evaluating the homomorphic `NOT` gate). Unfortunately, this defeats the benefits gained from asynchronous CUDA kernel launches and the CPU execution must block when it reaches a `NOT` gate until the GPU has finished evaluating all prior gates, instead of continuing to do more meaningful work. Similarly, nuFHE specializes in vectorized gates; for instance, it can evaluate a bitwise `AND` operation across 64-bit operands. However, this approach is very restrictive in terms of circuit evaluations as typically a circuit level is not composed of one type of gate.

REDcuFHE [29] enhances cuFHE to add multi-bit plaintext support and multi-GPU support. However, it still suffers from the same synchronization issues as cuFHE, and puts the burden of scheduling and handling communication between multiple GPUs on the programmer. ArctyrEX, on the other hand, handles all scheduling and communication procedures automatically.

Narisada et al. [49] introduce a methodology that relies on a redundant representation encoding of ciphertexts to accelerate large bit-width arithmetic operations for CGGI through GPU acceleration. As shown in Table 3, this work outperforms ArctyrEX for 32-bit addition and multiplication, but the number of implemented primitives are limited and bitwise operations are not supported. Additionally, the authors only provide evaluations for primitives and do not perform experiments with encrypted applications.

Lastly, Morshed et al. [46] present a GPU implementation of CGGI that leverages the NVIDIA cuFFT library and incorporates a set of handwritten circuits such as vector addition and matrix multiplication. Table 3 demonstrates that ArctyrEX outperforms [46] by a factor of about 1.5× for a small 32-bit addition and 16× for 32-bit multiplication (which is a significantly larger circuit). Additionally, ArctyrEX evaluates a vector addition with 32 elements of 32-bit integers 4.1× faster and a 16×16 matrix multiplication of 32-bit elements 10.6× faster. We also emphasize that all frameworks in Table 3, aside from ArctyrEX and the Google FHE Transpiler, require developers to write their own circuits by hand, as opposed to automatically generating them. As such, we only evaluate 32-bit arithmetic operations across all libraries.

A final class includes custom ASICs and FPGA implementations of FHE to act as a co-processor to speed up the underlying FHE arithmetic operations. However, many of these designs only support limited parameter sets [47, 60, 54, 55] and usually target other cryptosystems that enable approximate computing [56, 1] or computation with

modular integers [47]. On the other hand, ArctyrEX can support arbitrary parameters and implements the CGGI cryptosystem, which is more suitable for general-purpose computation.

8 Conclusion and Future Work

ArctyrEX is an end-to-end framework for encrypted computation that leverages GPU acceleration and incorporates novel strategies for executing FHE algorithms efficiently. For workloads such as neural network inference, we observe a linear speedup with increasing GPUs due to the inherent circuit-level parallelism, our proposed dispatch paradigm, and the high degree of primitive-level parallelism exploited by our CUDA-accelerated CGGI backend. Likewise, our evaluation using the AES-128 decryption benchmark shows that ArctyrEX can achieve faster runtimes compared to other contemporary frameworks based on CGGI, such as Concrete and the Goggle Transpiler. Even though hand-tuned frameworks (e.g., [59, 61, 62]) can achieve faster performance on specific benchmarks, our methodology is general-purpose with support for high-level C++ programs. As a result, our approach provides a more comprehensive solution: for example, while individual 32-bit arithmetic operations are not faster compared to [49], ArctyrEX offers more capabilities to end users, such as bitwise and comparison operations.

In future work, we plan to expand our frontend to support schemes beyond CGGI, as different schemes are better suited to different styles of computation. For instance, computing multiplications with large word sizes in CGGI is inefficient because the underlying circuit will be very large. Other schemes, like CKKS and BGV, support encrypting multi-bit values directly and can accomplish this multiplication in one primitive operation. Moreover, these schemes can achieve high throughput by operating on vectors of encrypted values. Indeed, the ArctyrEX frontend can be adapted for this use-case; after the HLS procedure, we are left with a high-level Verilog program that employs multi-bit operations (such as additions, multiplications, and multi-bit multiplexers). In the current implementation with CGGI, this Verilog is converted directly to a Boolean circuit through logic synthesis.

Acknowledgments

Authors acknowledge the members of Zama, CryptoLab, Google and Duality, particularly Ilaria Chillotti, Jung Hee Cheon, Ahmad Al Badawi, Yuri Polyakov, David Cousins, Shruti Gorantala and Eric Astor for their insightful discussions. We also acknowledge the constructive feedback from the anonymous reviewers and editors, which significantly contributed to the quality of this paper. C. Gouert and N.G. Tsoutsos would like to acknowledge the support of the National Science Foundation (Award 2239334).

References

[1] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi. Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 882–895, Los Alamitos, CA, USA, mar 2023. IEEE Computer Society.

[2] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Paper 2023/1040, 2023.

[3] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, page 53–63, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[5] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for mpc and fhe. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 430–454, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[6] Mazhar Ali, Samee U. Khan, and Athanasios V. Vasilakos. Security in cloud computing: Opportunities and challenges. *Information sciences*, 305:357–383, 2015.

[7] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 423–442, Cham, 2017. Springer International Publishing.

[8] Moran Baruch, Nir Drucker, Gilad Ezov, Yoav Goldberg, Eyal Kushnir, Jenny Lerner, Omri Soceanu, and Itamar Zimerman. Training large scale polynomial cnns for e2e inference over homomorphic encryption. *arXiv preprint arXiv:2304.14836*, 2023.

[9] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *WAHC '21*, page 57–62, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating homomorphic evaluation of deep learning predictions. In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning*, pages 212–230, Cham, 2019. Springer International Publishing.

[12] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 483–512, Cham, 2018. Springer International Publishing.

[13] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A compilation chain for privacy preserving applications. In *SCC '15*, page 13–19, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray CC 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, 2014.

[15] 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.

[16] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 347–368, Cham, 2019. Springer International Publishing.

[17] Eduardo Chielle, Oleg Mazonka, Homer Gamil, and Michail Maniatakos. Accelerating fully homomorphic encryption by bridging modular and bit-level arithmetic. In *ICCAD '22*, New York, NY, USA, 2022. Association for Computing Machinery.

[18] 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.

[19] 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*, pages 1–7, France (Virtual), Dec 2020. WAHC. Software library version: 2.7.0.

[20] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 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 5*, pages 1–19. Springer, 2021.

[21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 670–699. Springer, 2021.

[22] Jihoon Cho, Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 640–669, Cham, 2021. Springer International Publishing.

[23] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster CryptoNets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018. https://arxiv.org/pdf/1811.09953.

[24] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, Cambridge, MA, USA, 2022.

[25] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In Enes Pasalic and Lars R. Knudsen, editors, *Cryptography and Information Security in the Balkans*, pages 169–186, Cham, 2016. Springer International Publishing.

[26] Wei Dai and Berk Sunar. cufhe (v1.0). Available: https://github.com/vernamlab/cuFHE, 2018.

[27] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[28] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[29] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. REDsec: Running Encrypted Discretized Neural Networks in Seconds. In *Network and Distributed System Security Symposium (NDSS)*, pages 1–17, New York, NY, USA, 2023. IEEE Press.

[30] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09*, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[31] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *Public Key Cryptography – PKC 2012*, pages 1–16, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[32] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 850–867, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[33] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[34] Google Research. Google XLS. Available: http://google.github.io/xls/, May 2020.

[35] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, et al. A general purpose transpiler for fully homomorphic encryption. *arXiv preprint arXiv:2106.07893*, 2021. https://arxiv.org/pdf/2106.07893.

[36] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables. Cryptology ePrint Archive, Report 2023/1382, 2023. https://eprint.iacr.org/2023/1382.

[37] Charles Gouert and Nektarios Georgios Tsoutsos. Romeo: Conversion and evaluation of hdl designs in the encrypted domain. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, New York, NY, USA, 2020. IEEE Press.

[38] Charles Gouert, Mehmet Ugurbil, Dimitris Mouris, Miguel de Vega, and Nektarios Georgios Tsoutsos. Ripple: Accelerating Programmable Bootstraps for FHE with Wavelet Approximations. Cryptology ePrint Archive, Report 2024/866, 2024. https://eprint.iacr.org/2024/866.

[39] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 83–105, Cham, 2019. Springer International Publishing.

[40] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(1):9466–9471, Jul. 2019.

[41] Lei Jiang, Qian Lou, and Nrushad Joshi. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *DAC '22*, page 235–240, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Jae Hyung Ju, Jaiyoung Park, Jongmin Kim, Donghwan Kim, and Jung Ho Ahn. Neujeans: Private neural network inference with joint optimization of convolution and bootstrapping. *arXiv preprint arXiv:2312.04356*, 2023.

[43] Donghwan Kim, Jaiyoung Park, Jongmin Kim, Sangpyo Kim, and Jung Ho Ahn. Hyphen: A hybrid packing method and its optimizations for homomorphic encryption-based neural networks. *IEEE Access*, 2023.

[44] Dongwoo Kim and Cyril Guyot. Optimized privacy-preserving cnn inference with fully homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 18:2175–2187, 2023.

[45] Miran Kim, Xiaoqian Jiang, Kristin Lauter, Elkhan Ismayilzada, and Shayan Shams. Secure human action recognition by encrypted neural network inference. *Nature Communications*, 13(1):1–13, 2022.

[46] Toufique Morshed, Md Momin Al Aziz, and Noman Mohammed. CPU and GPU accelerated fully homomorphic encryption. In *International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 142–153, New York, NY, USA, 2020. IEEE, IEEE Press.

[47] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, Mizan Abraha Gebremichael, Homer Gamil, Eduardo Chielle, Ramesh Karri, Mihai Sanduleanu, and Michail Maniatakos. Cofhee: A co-processor for fully homomorphic encryption execution. In *2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–2, New York, NY, USA, 2023. IEEE Press.

[48] Vishnu Naresh Boddeti. Secure face matching using fully homomorphic encryption. In *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, pages 1–10, New York, NY, USA, 2018. IEEE Press.

[49] Shintaro Narisada, Hiroki Okada, Kazuhide Fukushima, Shinsaku Kiyomoto, and Takashi Nishide. Gpu acceleration of high-precision homomorphic computation utilizing redundant representation. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 1–9, 2023.

[50] NuCypher. nuFHE (v0.0.3). Available: https://github.com/nucypher/nufhe, 2019.

[51] NVIDIA. A100 Tensor Core GPU architecture. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[52] NVIDIA. CUDA, release: 12.1. Available: https://docs.nvidia.com/cuda/, 2023.

[53] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: Privacy-preserving, accurate and efficient inference for transformers. Cryptology ePrint Archive, Paper 2023/1893, 2023.

[54] 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, New York, NY, USA, 2019. IEEE Press.

[55] 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*, page 238–252, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 173–187, New York, NY, USA, 2022. Association for Computing Machinery.

[57] Ashish Singh and Kakali Chatterjee. Cloud Security Issues and Challenges: A Survey. *Journal of Network and Computer Applications*, 79:88–115, 2017.

[58] Roy Stracovsky, Rasoul Akhavan, and Florian Mahdavi Kerschbaum. Faster evaluation of aes using tfhe. In *FHE.org*, page Poster, 2022. https://rasoulam.github.io/data/poster-aes-tfhe.pdf.

[59] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. A homomorphic aes evaluation in less than 30 seconds by means of tfhe. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 79–90, 2023.

[60] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. Heaws: An accelerator for homomorphic encryption on the amazon aws fpga. *IEEE Transactions on Computers*, 69(8):1185–1196, 2020.

[61] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: faster and smaller. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 342–372. Springer, 2024.

[62] Benqiang Wei, Ruida Wang, Zhihao Li, Qinju Liu, and Xianhui Lu. Fregata: Faster homomorphic evaluation of aes via tfhe. In *International Conference on Information Security*, pages 392–412. Springer, 2023.

[63] Clifford Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/, 2014.

[64] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 19–35, USA, 2016. USENIX Association.

[65] Yatao Yang, Qilin Zhang, Wenbin Gao, Chenghao Fan, Qinyuan Shu, and Hang Yun. Design on Face Recognition System with Privacy Preservation Based on Homomorphic Encryption. *Wireless Personal Communications*, 123(4):3737–3754, 2022.

[66] Jiawen Zhang, Jian Liu, Xinpeng Yang, Yinghao Wang, Kejia Chen, Xiaoyang Hou, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. Cryptology ePrint Archive, Report 2024/136, 2024. https://eprint.iacr.org/2024/136.

[67] Mengxin Zheng, Qian Lou, Fan Chen, Lei Jiang, and Yongxin Zhu. Cryptolight: An electro-optical accelerator for fully homomorphic encryption. In *ACM International Symposium on Nanoscale Architectures*, pages 1–2, 2022.

[68] Itamar Zimerman, Moran Baruch, Nir Drucker, Gilad Ezov, Omri Soceanu, and Lior Wolf. Converting transformers to polynomial form for secure inference over homomorphic encryption. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235

of *Proceedings of Machine Learning Research*, pages 62803–62814. PMLR, 21–27 Jul 2024.

A Basic Linear Algebra Subroutines

The Basic Linear Algebra Subroutines are FHE benchmarks that form core components of algorithms in a wide variety of fields, such as image processing and machine learning. We focus on three distinct tensor multiplication algorithms on 16-bit encrypted data: a dot product of two vectors of length 500, a matrix-vector multiplication between a vector of length 125 and a 125×4 matrix, and a matrix multiplication between two 10×10 matrices. Additionally, we include a vector addition between two vectors of length 500; this benchmark was executed with a larger wordsize than the previous ones to increase its computational complexity. We compare a 256-thread CPU execution of these tensor algorithms with our approach running on up to 8 GPUs.

In Figure 11 and Figure 12, the dark-green bars show running time, and the light green bars plot the speedup of GPU vs. CPU. One A100 is 6.1× faster than the CPU reference running on the 256-threaded CPU execution model, and 8 A100s are 42.5× faster. We show the latency of these circuits for an increasing number of A100 GPUs and the speedup for all GPU configurations versus a CPU configuration with 256 threads. Our analysis shows a linear speedup with more GPUs, as our design exploits the ample circuit-level parallelism in both synthesis and runtime phases.

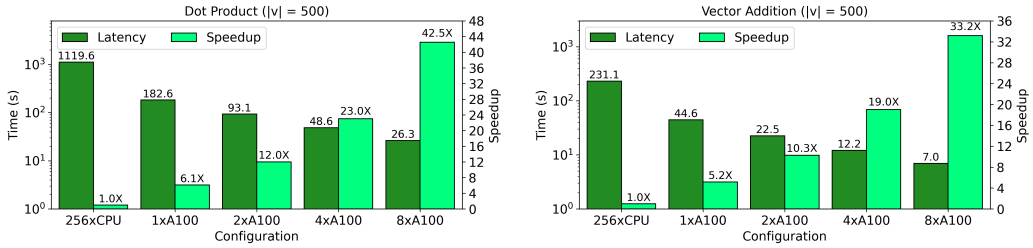


Figure 11: **Vector Algebra Benchmarks**: All dot products are performed with 16-bit encrypted elements and the vector addition is performed with 32-bit elements. The speedup bars are relative to the CPU implementation with 256 threads. $|v|$ indicates the vector length.

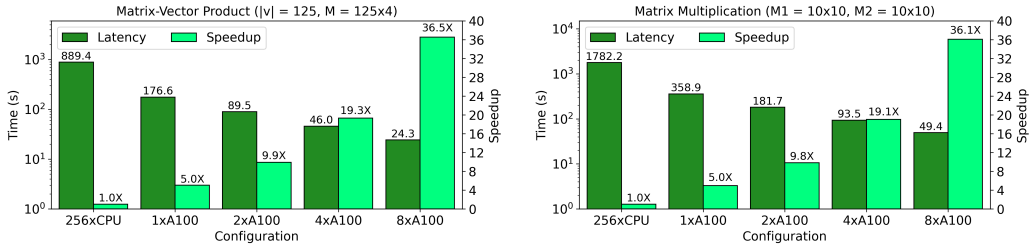


Figure 12: **Matrix Algebra Benchmarks**: All products use 16-bit encrypted elements. The speedup bars are relative to the CPU implementation with 256 threads. $|v|$ indicates the vector length and M refers to the dimensions of the matrices.

Figure 13 shows the linear algebra topologies; vector addition is more performant as the critical path is 2× shorter and the levels remain wide. This is reflected in the execution times in Figure 11, where the vector addition runs nearly 4× faster on 8 GPUs.

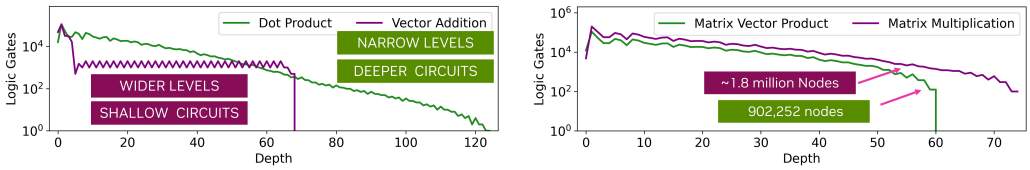


Figure 13: **Topology of Linear Algebra Benchmarks**: Vector addition is better suited for circuit for encrypted evaluation as it exhibits wide levels and a short critical path. The matrix-vector product and matrix multiplication benchmarks exhibit ample parallelism, with matrix multiplication being consistently twice as wide for most levels.

Additionally, we observe that Concrete can perform an identical vector addition in less than a second, with an overall latency reduction of 35.5× relative to ArctyrEX. We remark that this is primarily due to Concrete's ability to encode multi-bit values in a single ciphertext and natively supports addition as a primitive operation. On the other hand, ArctyrEX needs to utilize a Boolean adder to perform the equivalent operations. However, for the multiplication-intensive vector dot product, ArctyrEX is 29.3× faster than Concrete, which needs to perform costly programmable bootstraps to evaluate multi-bit multiplication. Both matrix benchmarks have very wide levels and are well-suited for evaluation on multi-GPU systems. For a matrix-vector product, ArctyrEX is 11.6× faster than Concrete and 14× faster for the 10×10 matrix multiplication.

B Transciphering Benchmarks

In line with the AES decryption circuit benchmark presented in Section 6, other ciphers can be utilized to accomplish transciphering and thus reduce the overall communication overhead between client and server. We explore two alternative constructions in the form of the Simon and Speck sister ciphers developed by the US National Security Agency [9]. Both are Fiestel ciphers, but Simon targets efficient hardware performance while Speck was designed with software in mind.

Table 4: Amortized decryption latency for Speck and Simon

Configuration	Speck Round (s)	Simon Round (s)
256xCPU	3.61	1.18
1xA100	0.59	0.21
2xA100	0.48	0.12

Table 4 presents the cost per round to evaluate Simon and Speck per 128-bit block size. Overall, Simon is more efficient than Speck because it uses strictly bitwise operations, whereas Speck has a 64-bit subtraction in each round that corresponds to a large Boolean circuit. This is expected as Simon was specifically designed for hardware deployment and therefore synthesizes to a smaller Boolean circuit.

C AI-Based Demonstrations with FHE

Many recent studies aim to address the challenges of privacy-preserving neural networks and transformer inference using different FHE schemes. For the former, both HyPHEN [43] and NeuJeans [42] (built on CKKS) can perform ImageNet classification with ResNet-18 in less than 15 seconds on an A100 GPU. On the other hand, Kim and Guyot [44]

evaluate CIFAR-100 inference with CKKS in 255 seconds with a single CPU thread. Baruch et al. [8] propose training strategies tuned for FHE evaluation and perform ResNet-18 inference in just under 7.5 minutes with an A100 GPU. Homomorphic transformer inference is a growing research area as a result of the popularity of services such as ChatGPT. Zimerman et al. [68] achieved secure inference over 128 tokens for a BERT-like transformer architecture in approximately 211 seconds with an A100 and AMD EPYC 7763 CPU. NEXUS [66] (CKKS-based) can evaluate BERT-base over 128 tokens in 37 seconds on 4 A100 GPUs and the deeper Llama-3-8B over 8 tokens in nearly 52 seconds. Lastly, BOLT [53] evaluates BERT-base in 91 seconds with the BFV cryptosystem in a leveled context with 2 `c6i.16xlarge` AWS instances.