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, 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 over 40× faster on an NVIDIA DGX A100 and 6× faster with a single A100 relative to a 256-threaded CPU baseline.

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 can provide 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 each can contain a wealth of sensitive data from multiple users [47] [4] [50]. FHE allows users to encrypt their data locally, outsource the resulting ciphertexts to the cloud, have the cloud evaluate meaningful algorithms on the encrypted data, receive the encrypted data after processing, and decrypt the result to get the final output. This can be used for a wide variety of applications, such as privacy-preserving machine learning as a service (MLaaS) [28] [21] [14] and facial recognition [52] [7].

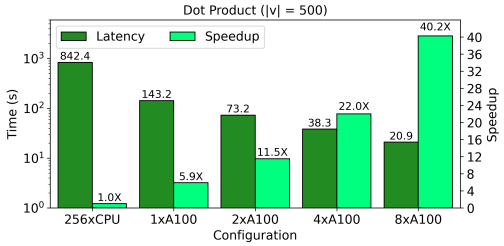
FHE was realized in 2009 with the advent of the bootstrapping procedure which allows unlimited computation on ciphertexts [29]. However, early FHE was plagued by both high memory requirements and enormous computational overheads, which rendered it infeasible for adoption. Since its inception, great strides have been made to reduce these runtime costs: First, new homomorphic encryption schemes have been developed with more efficient bootstrapping constructions, such as DM [25] and CGGI [18]. Additionally,

```
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;
}
```

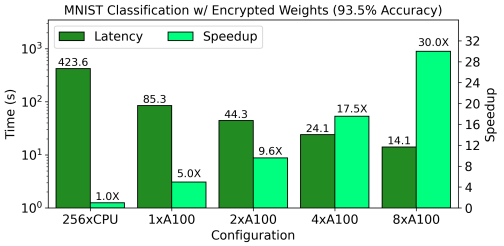
(a) Dot Product Code

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

(b) Fully-Connected Layer Code



(c) Dot Product Performance



(d) MNIST Classification Performance

Figure 1: **Practical Large-Scale Applications**: Using our approach, a dot product subroutine runs $\approx 6\times$ faster on a single A100 GPU and over $40\times$ faster with an NVIDIA DGX A100 relative to a multi-threaded CPU execution with 256 threads, resulting in an end-to-end application level speed up of $30\times$ for MNIST classification.

various algorithmic and software optimizations, such as HE-friendly number theoretic transforms (NTT) [23], 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 [34] [15] [5]. Lastly, CPU-based acceleration techniques were also adopted, including AVX and FMA extensions [8]. However, the algorithmic level performance gains have recently stagnated and further speedups are coming only from hardware acceleration.

The most prominent hardware platforms for encrypted computation with FHE are GPUs, which have been thoroughly demonstrated to be particularly suited for the types of arithmetic required by modern FHE constructions. Most encrypted operations expose ample parallelism and are computationally intensive [41]; 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 [44] and cuFHE [24] 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 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 up to 40× faster on 8 GPUs;

- A new optimized backend for the CGGI cryptosystem that prioritizes fast evaluation of arbitrary algorithms and outperforms state-of-the-art implementations by more than an order of magnitude. This enables 32-element vector addition of 32-bit integers up to 4.1× faster, and 16×16 matrix multiplication of 32-bit elements up to 10.6× faster on 8 GPUs.

Our proposed framework makes large-scale applications practical in FHE. Figure 1(a) showcases the high-level input code used to run both a large dot product of two 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 technique called *modulus switching*, with larger encryption parameters allowing higher noise tolerance. However, larger parameters entail slower computation and higher memory consumption, which limits 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 [29], *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 [30]. Therefore, the only way to achieve feasible FHE for general-purpose computation is to accelerate and optimize the bootstrapping mechanism.

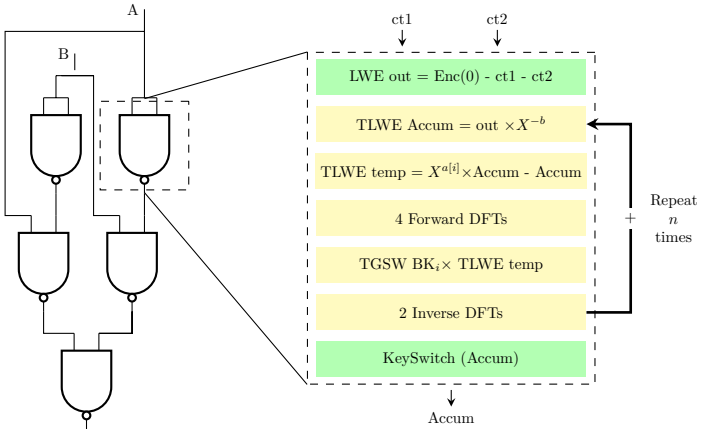


Figure 2: **Encrypted Logic Gate Evaluation**: All standard two-input logic gates begin with a series of linear operations between the input LWE ciphertexts, followed by a bootstrapping procedure (executed by the looped instructions), and lastly a keyswitching operation. The steps in yellow represent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

The CGGI Cryptosystem

Both the DM [25] and CGGI cryptosystems [18] possess bootstrapping mechanisms that can be evaluated on the order of tens of milliseconds on a CPU using modern open-source implementations such as Concrete [19] and OpenFHE [1], which is much 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 decrease/reset the amount of noise in the ciphertext. In the case of CGGI, which evaluates Boolean gates, bootstrapping must be performed after 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 each contains nearly 25,000 encrypted logic gates. Evaluating this circuit results in over 75 million invocations of the DFT. DM [25] was the first cryptosystem to introduce a *functional bootstrap* that can refresh ciphertext 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 multiplexer that is capable of obliviously choosing between two encrypted bits dependent on the underlying value of an encrypted selector bit. For all gates except for the trivial inverter gate, which is noiseless and composed of strictly linear operations, the bootstrapping operation comprises the majority of the gate's latency [36]. In turn, the core bottleneck of bootstrapping is the numerous polynomial multiplications between the encrypted secret key components and input ciphertexts. All state-of-the-art FHE libraries opt to perform these high-degree polynomial multiplications as element-wise multiplications in the DFT domain, which is asymptotically faster than textbook polynomial multiplication [13] [22]. 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, the FFT requires additional type conversions between integers and floating point numbers as FHE ciphertexts contain strictly integer coefficients. As a result, the 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 LWE ciphertexts, serve as the inputs and outputs of each homomorphic gate evaluation from a user perspective. LWE 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 TLWE ciphertexts (i.e., Ring-LWE) that are larger in size. Typically, TLWE 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 TLWE 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 TLWE 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).

Overall, the CGGI cryptosystem is a good candidate for achieving accelerated general purpose computation on GPUs for a variety of reasons. First, the parameters used by CGGI are often significantly smaller than other FHE schemes, which yields smaller ciphertexts. Thus, multiple ciphertexts can be held in the GPU shared memory simultaneously, which is not always the case for schemes such as CKKS [16], BFV [26], and BGV [11] that can utilize ciphertexts on the order of several megabytes [48]. 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 [9] [35]. Lastly, 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. 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 and therefore this approach is better suited for specific applications rather than for arbitrary computation. 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.

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 is depicted in Figure 3, illustrating the capabilities of the frontend, runtime schedule coordination, and backend operations. 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 the synthesis flow used to convert input programs to FHE code for outsourced computation.

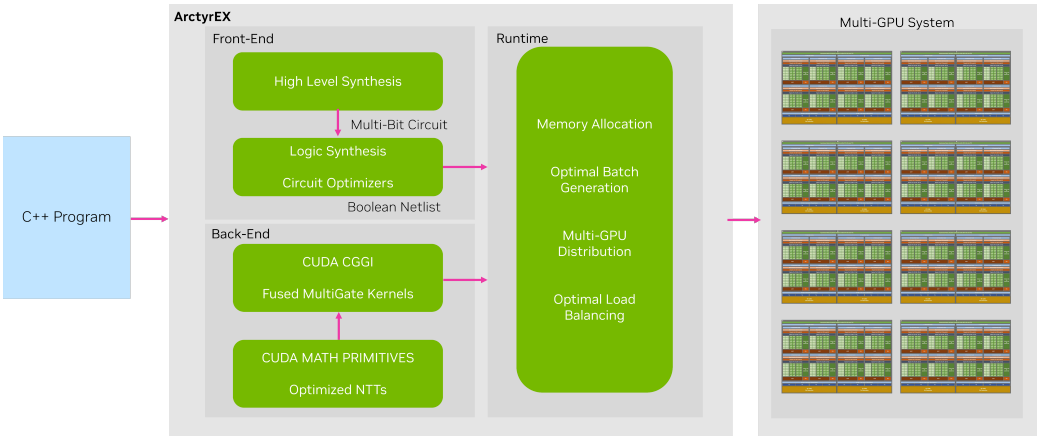


Figure 3: **System Overview:** Our proposed system is composed of three distinct layers that work together to realize an end-to-end framework for scalable encrypted computation. The frontend converts high-level programs to a logic circuit tuned for FHE. In turn, this logic circuit is parsed by the middle 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.

Optimal FHE Circuit Characteristics

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 kernels of 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 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 don't 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.

```c
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];
      }
    }
  }
}
```

```c
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 approximation: 40320 + 20160*x - 1680*x^3 + 168*x^5 - 17*x^7
  int temp = 40320;
  int temp_2 = 20160 * product;
  int squared = product * product;
  temp = temp + temp_2;
  temp_2 = squared * product * 1680;
  temp = temp - temp_2;
  squared = squared * squared;
  temp_2 = squared * product * 168;
  temp = temp + temp_2;
  squared = squared * product;
  squared = squared * product;
  temp_2 = squared * 17;
  product = temp - temp_2;

  // Client post-processes classification score by dividing by 80640
  return product;
}
```

Figure 4: High Level Synthesis Kernels for General Matrix to Matrix Multiplication (GEMM) and Logistic Regression (LR) Inference.

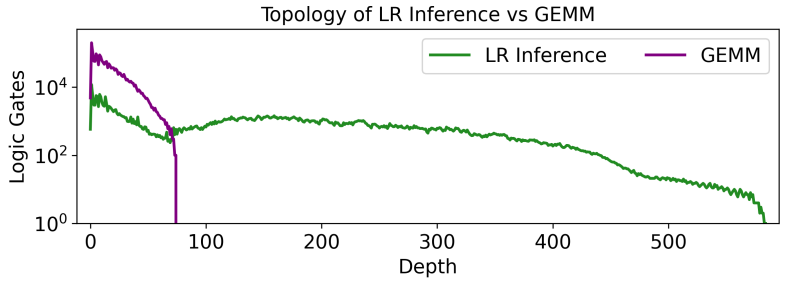


Figure 5: **Circuit-level parallelism**: Visualizing the circuit topology of GEMM versus LR inference illustrates important differences; the GEMM circuit is ideal for parallel evaluation, while LR is less suitable.

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 [51], 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 [49]. 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 [32], 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 [32, 33], 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 provided 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. In fact, a 10×10 matrix multiplication application generates several hundred thousand lines of code and the GCC compiler is unable to generate an executable for this large program. Moreover, when code is generated in this fashion and the gate invocations are dumped one after another into the output program, it is impossible to parallelize the gates as there is no intuition which gates can be evaluated concurrently.

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 [32] 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.

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 [51]. Each node contains an opcode, which defines the operation performed by the corresponding 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 boolean values, and we augment the IR generated by the frontend with gates which copy an input ciphertext into another one. These copies are, for example, used to integrate the retrieval of encrypted results as part of the circuit itself instead of having to extract individual ciphertexts after waiting for the termination of the circuit.

To derive a parallel execution of the circuit, we first dispatch all gates into multiple *waves*. Each wave must be processed in-order, but all entries of a *wave* can be processed concurrently. We now detail 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 the 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. It contains at least an entry because there must exist a gate which does not depend on other gates. Until the FIFO is empty, we remove the first entry n from the FIFO, and do the following :

- We assign n to the first wave if there are no input dependencies, or we compute the maximum index of the wave of all predecessors, and add n to the next wave. All predecessors have been assigned an index, otherwise 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 the indexes of their predecessors, all entries in a wave are independent

and can be processed concurrently, as long as the different 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. In order to use multiple processing units, we dispatch waves over the different devices. A simple solution to dispatch a wave with N gates over K devices which consists of splitting it into roughly N/K gates per device, as illustrated in Figure 6.

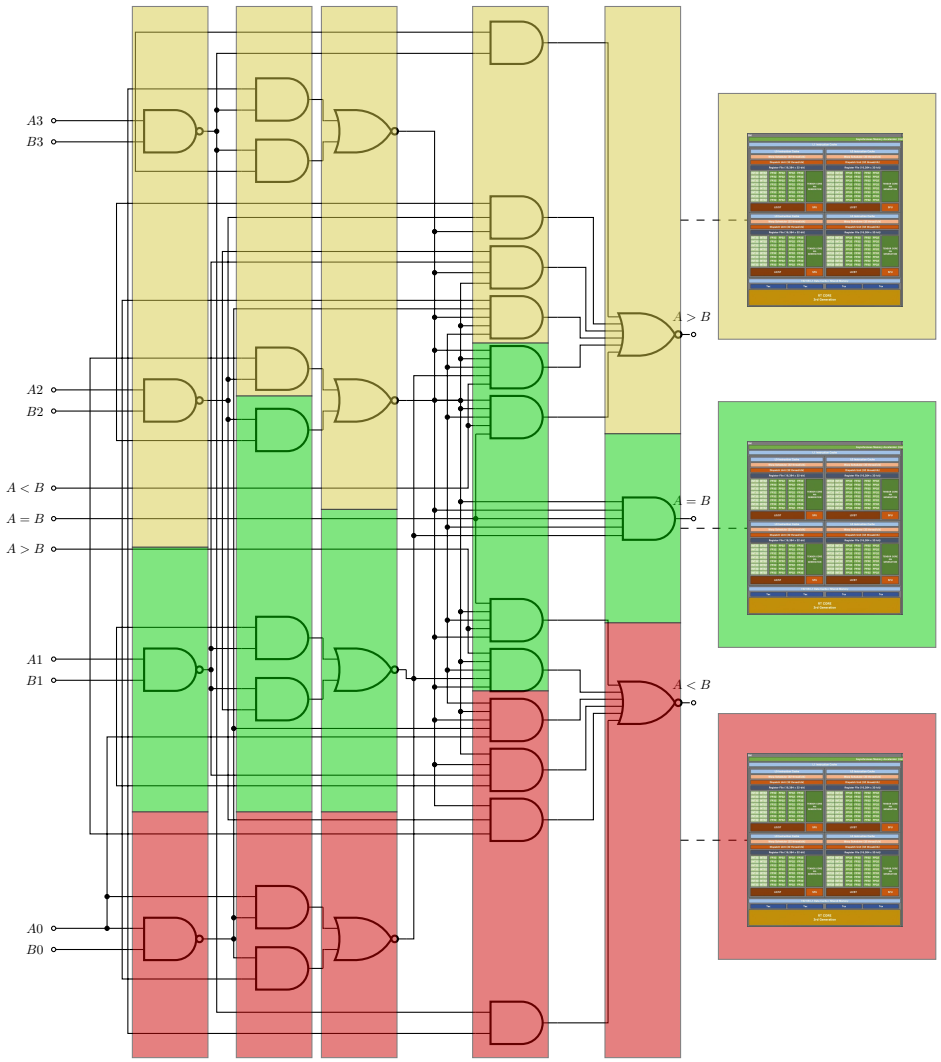


Figure 6: **Mapping gates to devices**: Circuit gates are dispatched into independent waves, which are then split across the different processing units. In this example, we extract 5 waves which are spread over 3 devices that receive a similar workload.

Let us consider a wave with 43875 gates, composed of 2125 `AND` gates, 25000 `OR` gates and 16750 `NOT` gates. On 2 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 number of gates between the different devices is therefore not a satisfactory approach, and only results in a 1.7x speedup with only 2 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 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 a 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 different processing units. This section details our native CUDA implementation of the CGGI cryptosystem, 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 [45].

Bootstrapping keys have a relatively large size, of approximately 100 MB for 110 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 though 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 cipertexts 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 will consume an assigned batch from the producer and outsource the computation to a dedicated GPU. This approach is quite 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.

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 introduce 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, 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. We therefore 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 [24] or allow for "vectorized" gates (such as performing a bitwise NAND between two 32-bit ciphertext arrays) [44]. 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 a single 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 the user 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 [46].

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 bootstrapping in CGGI is largely determined by the efficiency of the DFT used to facilitate polynomial multiplication. Both nuFHE [44] and cuFHE [24] use the NTT for this operation, and both employ the same general 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 the modular reduction step 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

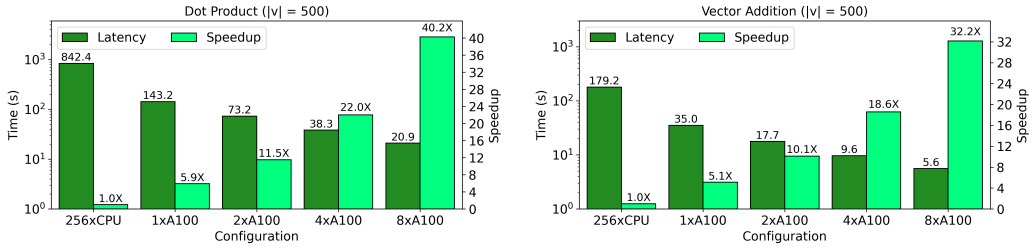


Figure 7: **Vector Algebra Benchmarks**: All dot products are performed with 16-bit encrypted elements and the vector addition is performed with 32-bit elements. The speedup bars are relative to the CPU implementation with 256 threads. $|v|$ indicates the vector length and M refers to the dimensions of the matrices.

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). Unless otherwise indicated, all benchmarks were run with parameters corresponding to 110 bits of security based on the BKZ-beta classical cost model provided by the state-of-the-art LWE estimator framework [2]. Specifically, for RLWE ciphertexts used in bootstrapping, we utilize a ring dimension of 1024 and set the noise rate to 25×10^{-9}. These are the same RLWE parameters employed by the TFHE library [18] for their parameter set corresponding to 128 bits of security. For LWE ciphertexts, we utilize $n = 512$ and a noise rate of 2^{-15}, which yields approximately 110 bits of security. As such, the overall security of the parameter set

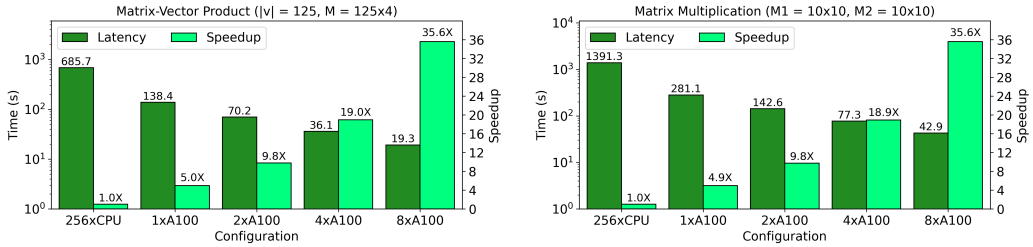


Figure 8: **Matrix Algebra Benchmarks**: All products are performed with 16-bit encrypted elements. The speedup bars are relative to the CPU implementation with 256 threads. $|v|$ indicates the vector length and M refers to the dimensions of the matrices.

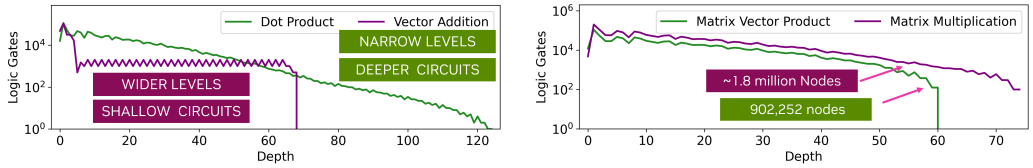


Figure 9: **Topology of Linear Algebra Benchmarks**: Vector addition is better suited for circuit for encrypted evaluation as it exhibits wide levels and a short critical path. The matrix-vector product and matrix multiplication benchmarks exhibit ample parallelism, with matrix multiplication being consistently twice as wide for most levels.

used for the following experiments is 110 bits of security. This cost model is also employed by the TFHE library in the security analysis of its hard-coded parameter sets [18].

FHE Basic Linear Algebra Subroutines

The FHE Basic Linear Algebra Subroutines are 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 7 and Figure 8, the dark-green vertical bars show running time, and the light green vertical bars plot the speedup of the GPU execution vs. the CPU execution. One A100 is 5.9× faster than the reference implementation running on the 256-threaded CPU execution model, and 8 A100s are 40× 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 by increasing numbers of GPUs, as our design exploits the ample circuit-level parallelism in both synthesis and runtime phases.

Figure 9 depicts the width of each level in the linear algebra benchmarks; the vector addition is more performant as the critical path is approximately 2× shorter and the levels remain relatively wide, increasing parallelism opportunities. Indeed, this is reflected in the execution times in Figure 7, where the vector addition runs nearly 4× faster on 8 GPUs. Both matrix benchmarks have very wide levels and are well-suited for evaluation on multi-GPU systems.

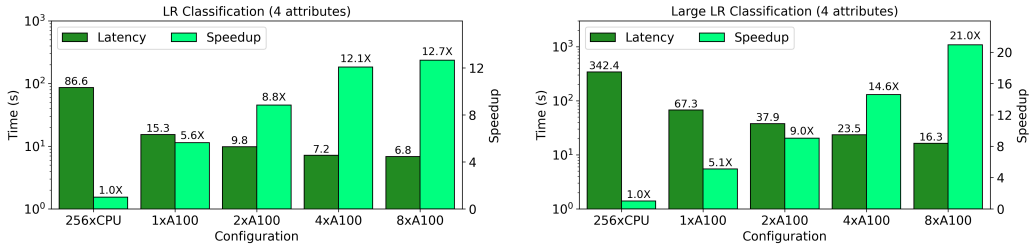


Figure 10: **Logistic Regression Inference**: We employ 32-bit words for the small approximation and a 64-bit words for the large approximation to avoid overflow. We observe a better scaling trend for the higher accuracy LR because it exhibits wider levels.

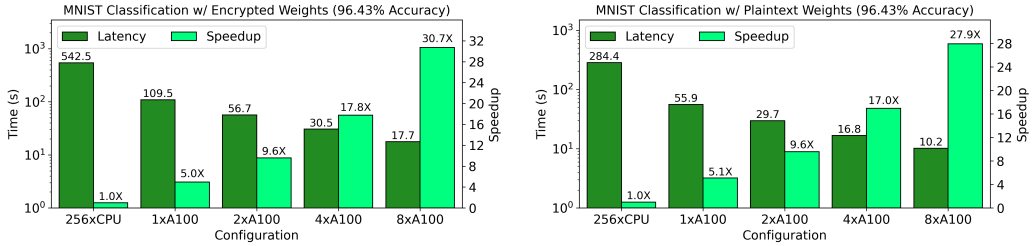


Figure 11: **Neural Network Inference**: The encrypted weight variants for both network configurations represent the scenario where the computing party does not own the model. On the other hand, the variants with plaintext weights correspond to the scenario where the computing party owns the proprietary model. We observe a roughly 2× speedup when plaintext weights are used.

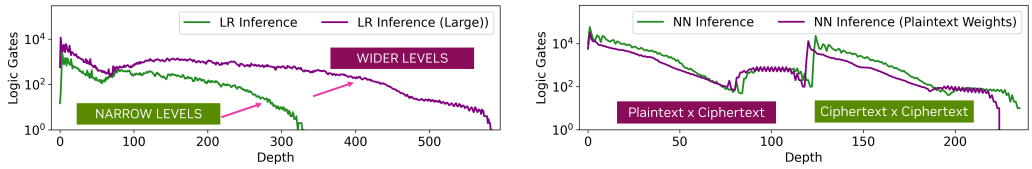


Figure 12: **Topology of Machine Learning Benchmarks**: For LR inference, the large variant uses a more accurate sigmoid approximation. It is much deeper due to a larger word size and more polynomial terms evaluated. The neural network plaintext weight variant exhibits a shorter critical path and is composed of much fewer gates overall.

Encrypted Machine Learning Applications

One of the most widely explored use-cases for FHE is privacy-preserving machine learning as a service. This paradigm can be divided into two distinct scenarios depending on who owns the ML model. In its most basic form (first scenario), a client with sensitive inputs wants to have them classified but does not have either the computational resources or trained network to do so. The client can encrypt their data homomorphically, upload to a third-party cloud server, and receive encrypted classification results after the cloud server computes the inference procedure.

The two scenarios essentially differ depending on who owns the proprietary model. If the cloud is the owner, it can simply perform the inference procedure with encrypted inputs and cleartext weights and biases. This results in faster FHE operations overall, as plaintext-ciphertext operations are considerably less costly than ciphertext-ciphertext operations. Even though the model parameters are in plaintext form, there is no security concern as long as they never leave the cloud server itself. The second scenario involves

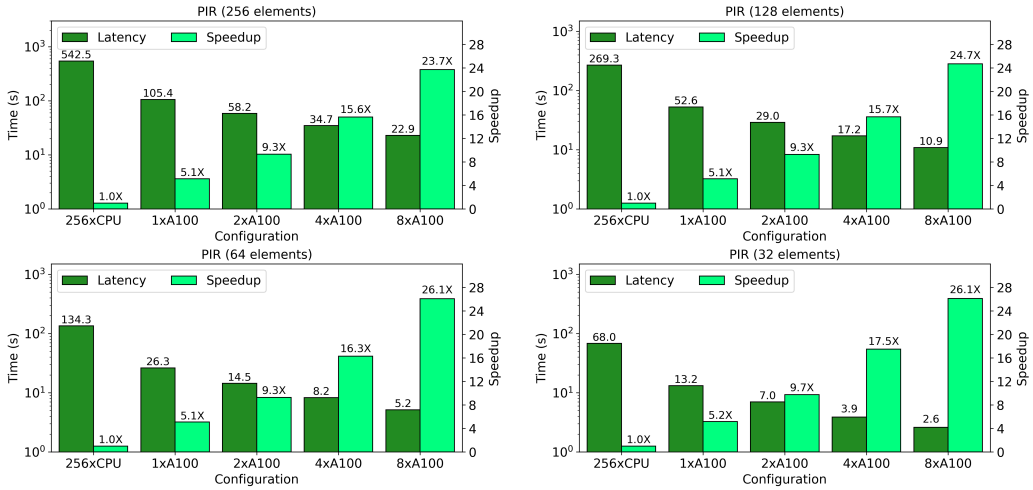


Figure 13: **Private Information Retrieval**: The reported time outlines the cost of a query over an encrypted key-value storage with 64-bit encrypted keys and values.

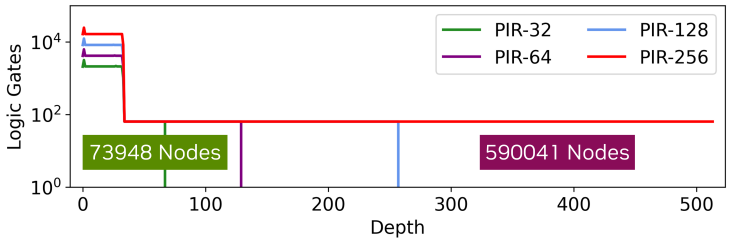


Figure 14: **Private Information Retrieval Topology**: Increasing the size of the database linearly increases the critical path, and widens the first few layers of the circuit.

either the client or an additional party owning the proprietary ML model. In this case, the client or model owner will encrypt the weights and biases homomorphically, and upload them to the cloud server prior to the inference procedure. Now, all computations are strictly ciphertext-ciphertext operations, and the overall inference cost is more expensive. The majority of existing works demonstrating ML inference with FHE adopt the first scenario as it is generally much more efficient [10, 21, 28].

Our analysis considers two important machine learning procedures for encrypted classification in the form of logistic regression (LR) inference and neural network (NN) inference. For logistic regression, we perform binary classification for datasets with four attributes, such as the Iris dataset [27]. 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; the more terms that are evaluated, the more accurate and more computationally expensive the approximation becomes. 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 10 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 12. Using 8 GPUs with the more accurate sigmoid approximation still outperforms the CPU implementation with 256 threads by 21×.

For neural network inference, we employ the same network architecture used by FHE-DiNN [10]. The network consists of two fully-connected layers with a sign activation function and is used to classify the MNIST dataset of handwritten digits. 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 11 while Figure 12 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. However, this variant also exhibits lower scaling for large numbers of GPUs because less computation is involved.

Cryptography and Security Applications

Transciphering

At first glance, it seems odd to compute an encryption algorithm homomorphically when the data is already encrypted. However, these algorithms enable an exciting strategy called *transciphering* that dramatically reduces the large communication overhead associated with FHE [3, 20]. Instead of sending large homomorphic ciphertexts to the cloud for outsourced computation, 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 110 bits of security, this strategy can decrease the communication overhead associated with the client sending encrypted inputs by a factor of over 16000×.

Table 1: Amortized cost of decryption rounds for Speck and Simon

Configuration	Speck-128/128 Round (s)	Simon-128/128 Round (s)
256xCPU	2.41	0.80
1xA100	0.34	0.13
2xA100	0.29	0.07

Our analysis employs the lightweight Simon and Speck ciphers proposed by the US National Security Agency [6]. These ciphers are well-suited to evaluate CGGI cryptosystems because they are primarily composed of bitwise operations. Other ciphers like AES are less suitable as they require expensive lookup-table evaluations, or a high number of finite-field arithmetic operations [31]. For both ciphers we use their 128/128 bit variants, as symmetric security needs to be commensurate to our FHE parameters. Table 1 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.

Private Information Retrieval

Aside from machine learning and cryptographic benchmarks, we explore another realistic and useful application of FHE enabled by ArctyrEX in the form of private information retrieval (PIR). The ability to search and perform computation across an encrypted database has many useful applications, such as managing a directory of health-care records that must be kept confidential for compliance with standards such as HIPAA [53]. We represent the encrypted database as a key/value storage where both keys and values are encrypted. Figure 14 demonstrates the circuit characteristics of a single query for databases of increasing size. On Figure 13, we observe a linear scaling with increasing

database size; this is expected as a query requires comparison operations with each record in the database due to the *termination problem*. Specifically, the termination problem states that it is impossible to make a decision based on encrypted data as the computing party does not know the underlying value of the ciphertext [43]. As such, each element of the database must be visited and the correct entry needs to be chosen through oblivious encrypted multiplexing operations. We also performed experiments across PIR problem sizes: Each value is 64-bits in size and we demonstrate the scalability of PIR with four database sizes (32, 64, 128, and 256 entries).

7 Related Works

Prior works can be divided into two categories: FHE compilers for general-purpose computation and acceleration 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.

Comparisons with State-of-the-Art FHE Compilers

The Cingulata framework (formerly Armadillo [12]) allows users to map C++ code into a sequence of `AND` and `XOR` gates. Cingulata works strictly 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 execution for CGGI and does not offer GPU support. 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 custom encrypted data types for leveraging FHE in general applications [17]. It supports a variety of cryptographic 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 [32] and Romeo [33] 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.

Comparisons with FHE Acceleration Frameworks

The cuFHE [24] and nuFHE [44] constitute the current state-of-the-art for GPU acceleration of the CGGI cryptosystem. The former is a proof-of-concept library that implements high throughput logic gate evaluations on a single NVIDIA GPU. However, cuFHE is

Table 2: Latency comparisons with existing backends for 32-bit arithmetic operations (taken from Morshed et al. [42]). ArctyrEX is at least 6.4× faster for 32-bit multiplication compared to prior works. Moreover, ArctyrEX evaluates vector addition with 32 elements 4.1× faster, and 16 × 16 matrix multiplication of 32-bit elements 10.6× faster.

Library	Security Level (bits)	Addition (s)	Multiplication (s)
ArctyrEX	**110**	**1.33**	**2.13**
FHE Transpiler [32]	80	6.53	13.56
Morshed et al. [42]	80	1.47	25.13
TFHE [18]	80	7.04	489.93
nuFHE [44]	80	3.08	137.78
cuFHE [24]	80	1.50	97.5

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 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 [28] 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. Lastly, Morshed et al. [42] present a GPU implementation of CGGI that leverage the NVIDIA cuFFT library and incorporates a set of handwritten circuits such as vector addition and matrix multiplication. Table 2 demonstrates that ArctyrEX outperforms [42] 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 2, aside from ArctyrEX and the Google FHE Transpiler, require developers to write their own circuits by hand, as opposed to automatically generating them.

8 Concluding Remarks and Future Work

ArctyrEX is the first end-to-end framework for general-purpose encrypted computation that leverages GPU acceleration and provides novel strategies for executing FHE algorithms efficiently on GPU-based HPC systems. For realistic workloads such as neural network inference, we observe a linear speedup with increasing numbers of GPU devices thanks to the inherent circuit-level parallelism, the proposed dispatch paradigm, and the high degree of primitive-level parallelism exploited by our CUDA-accelerated CGGI backend.

In future work, we plan to expand our frontend support to schemes beyond CGGI, as different schemes are better suited to different styles of computation, which can help achieve

higher throughput and high-accuracy encrypted deep learning inference and training. For instance, computing multiplications with large word sizes in CGGI is inefficient because the underlying circuit could eventually grow very large. Other schemes, like CKKS and BGV, support encrypting multi-bit values directly and can accomplish this multiplication in one primitive operation instead of many. Moreover, CKKS is an attractive option for certain machine learning applications, as it natively supports operations on encrypted floating-point values. With small modifications to our current ArctyrEX frontend, namely omitting the logic synthesis step, we can readily support all other FHE schemes that take the form of a general arithmetic circuit as opposed to purely Boolean circuits.

Developments in our middleware layer involves investigating further scheduling optimizations that further reduce device-to-device data transfers. A potential solution to this challenge involves incorporating graph partitioning methodologies to minimize inter-level dependencies between computing devices. Regarding our backend, future work will investigate alternative techniques to accelerate the DFT step, such as exploring further NTT acceleration on GPUs, as well as adopting the FFT. We also plan to investigate fusing gate evaluations across GPU streaming multiprocessors to minimize latency of FHE gates. This capability will be useful for thin circuit levels where the total number of gates is less than that of the total number of SMs across all available GPUs.

For the application level FHE optimization, future research involves integrating deep neural network optimizations such as [37, 39] and its correctness emphasis [38, 40] with optimizations in ArctyrEX frontend to achieve higher throughput and reliably accurate encrypted deep learning inference.

References

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, 2022.

[2] 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.

[3] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454. Springer, 2015.

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

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

[6] 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*, pages 1–6, 2015.

[7] 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. IEEE, 2018.

[8] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. Intel hexl: accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–62, 2021.

[9] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating homomorphic evaluation of deep learning predictions. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 212–230. Springer, 2019.

[10] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[12] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, pages 13–19, 2015.

[13] 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.

[14] 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.

[15] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2019.

[16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.

[17] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E3: A framework for compiling c++ programs with encrypted operands. *Cryptology ePrint Archive*, 2018.

[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*, volume 15, 2020.

[20] Jihoon Cho, Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 640–669. Springer, 2021.

[21] 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.

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

[23] Wei Dai and Berk Sunar. cuHE: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, pages 169–186. Springer, 2015.

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

[25] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.

[26] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

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

[28] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. REDsec: Running Encrypted Discretized Neural Networks in Seconds. *Cryptology ePrint Archive*, 2021.

[29] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[30] Craig Gentry, Shai Halevi, and Nigel P Smart. Better bootstrapping in fully homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–16. Springer, 2012.

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

[32] 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.

[33] 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. IEEE, 2020.

[34] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019.

[35] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 9466–9471, 2019.

[36] Lei Jiang, Qian Lou, and Nrushad Joshi. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. *arXiv preprint arXiv:2202.08814*, 2022.

[37] Vinu Joseph. *Programmable Neural Network Compression with Correctness Emphasis*. PhD thesis, University of Utah, USA, 2021.

[38] Vinu Joseph, Nithin Chalapathi, Aditya Bhaskara, Ganesh Gopalakrishnan, Pavel Panchekha, and Mu Zhang. Correctness-preserving compression of datasets and neural network models. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 1–9, 2020.

[39] Vinu Joseph, Ganesh Gopalakrishnan, Saurav Muralidharan, Michael Garland, and Animesh Garg. A programmable approach to neural network compression. *IEEE Micro*, 40(5):17–25, 2020.

[40] Vinu Joseph, Shoaib Ahmed Siddiqui, Aditya Bhaskara, Ganesh Gopalakrishnan, Saurav Muralidharan, Michael Garland, Sheraz Ahmed, and Andreas Dengel. Going beyond classification accuracy metrics in model compression. *arXiv preprint arXiv:2012.01604*, 2020.

[41] 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.

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

[43] Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Terminator suite: Benchmarking privacy-preserving architectures. *IEEE Computer Architecture Letters*, 17(2):122–125, 2018.

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

[45] NVIDIA. A100 Tensor Core GPU architecture, 2020.

[46] NVIDIA. CUDA, release: 12.1, 2023.

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

[48] Mayank Varia, Sophia Yakoubov, and Yang Yang. HEtest: A homomorphic encryption testing framework. In *International Conference on Financial Cryptography and Data Security*, pages 213–230. Springer, 2015.

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

[50] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *25th USENIX security symposium (USENIX Security '16)*, pages 19–35, 2016.

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

[52] 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.

[53] Rui Zhang, Rui Xue, and Ling Liu. Searchable encryption for healthcare clouds: a survey. *IEEE Transactions on Services Computing*, 11(6):978–996, 2017.

A Developing FHE Applications

ArctyrEX is designed to run efficiently for any algorithm, yet the way that an algorithm is expressed can have an impact on the HLS procedures that map the program to a Boolean netlist. The most important consideration lies in the complexity of the high-level application code. For complex algorithms that contain several thousand loop iterations or large loop bodies, the time required to perform the HLS and logic synthesis can increase significantly, or the synthesis toolchain itself may be unable to successfully generate a circuit. Notably, we can overcome this challenge by splitting an algorithm into multiple HLS inputs and invoking them one after the other in the encrypted application. Figure 15 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 GEMM procedure.

```
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 15: Partial Matrix Multiplication HLS Kernel

B NVIDIA DGX A100 System

The NVIDIA A100 DGX used in this work consists of 8 A100 GPUs. Two distinct groups of four GPUs are inter-connected using high speed NVLink buses. These GPUs have hardware support for direct access to *registered* host memory, which we leverage for intermediate encrypted wire transfers before they are cached into the shared memory of the devices during gate evaluation. Along with the synchronization mechanisms employed by our custom dispatching system, this naturally ensures data consistency across multiple devices. The NVIDIA A100 GPUs used in our experimental evaluation are data center GPUs, which is consistent with prior works (e.g., [19]). Each GPU has 108 streaming multiprocessors that act as independent processing units. Each streaming multiprocessor can evaluate a logic gate in the context of ArctyrEX, allowing for 108 concurrent gate evaluations on a single GPU.

C Security Considerations and Threat Model

ArctyrEX generates code 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 has incentives to view 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 bases its security on the (R)LWE problems. In cryptography, the security of a cipher is established using cryptanalysis and the security is derived from a reliance on underlying mathematical problems that are known to be NP-hard. this is directly applicable to CGGI, as LWE and its variants are all hard lattice problems.