Towards a Scalable Multi-GPU System for Encrypted AI

Joshua Kim

May 2025

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Advisor: Prof. Dimitrios Skarlatos

Submitted in partial fulfillment of the requirements
for the SCS Honors Undergraduate Research Thesis.

Keywords: Fully Homomorphic Encryption, GPU, Encrypted AI, Parallelism

For my family and friends, who have supported me through my journey at CMU.

Abstract

Fully homomorphic encryption (FHE) is a cryptographic scheme that offers user data privacy by supporting computations directly on encrypted data. However, it suffers from a severe performance overhead of four orders of magnitude compared to plaintext computations. In this work, we present Caramel, a GPU-targeting acceleration backend and a set of kernel fusion techniques for accelerating FHE-based workloads. Caramel aims to minimize overheads unique to GPU implementations of FHE, specifically addressing kernel launch and memory inefficiencies. It also introduces novel usage of float compute units on GPUs for FHE, which is composed of modular arithmetic operations. Caramel leverages parallelism within FHE workloads by utilizing multi-GPU systems to enhance performance and scalability. The results demonstrate that Caramel addresses kernel launch overheads and displays good scalability in our parallelism model. Finally, it presents clear design paths to make FHE practical in multi-GPU environments.

Acknowledgments

Thank you to my advisor Dimitrios Skarlatos for his immense guidance and patience, and to Siddharth Jayashankar for all his support throughout the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing and datacenters have become the foundation of many of the services that we rely on today, including healthcare, finance, and other multi-application domains. This large scale usage of the cloud has greatly increased the user data being processed. When data is utilized in cloud computing, it is encrypted at rest and in transport. However, it must be decrypted at use, potentially revealing sensitive data. This raises various privacy concerns, where this data is now vulnerable to various attacks such as data breaches and side channels. Unfortunately, this is not just a theoretical concern, as we've seen large scale attacks take place [8, 28]. This concern has led to restrictive regulations that threaten to limit the use of private data - case in point, Italy's temporary ban on ChatGPT [19]. Consequently, it is becoming increasingly important to identify computation methods that will never reveal sensitive data to external parties.

Fully homomorphic encryption (FHE) [11] is a cryptographic scheme that allows direct computation on *encrypted data*. Figure 1.1 illustrates the security model of FHE on the cloud. A client can upload encrypted data `Enc(x)`, on which models can apply operations f, and the client can decrypt the result with the same key, obtaining `f(x)= Dec(f(Enc(x)))`. Of the various benefits that FHE offers, the most promising application is privacy-ensured machine learning tasks. However, FHE is currently impractical in real-world applications, as it suffers from a severe performance overhead of 4 orders of magnitude on state-of-the-art CPU FHE designs [2, 25] compared to equivalent plaintext workloads.

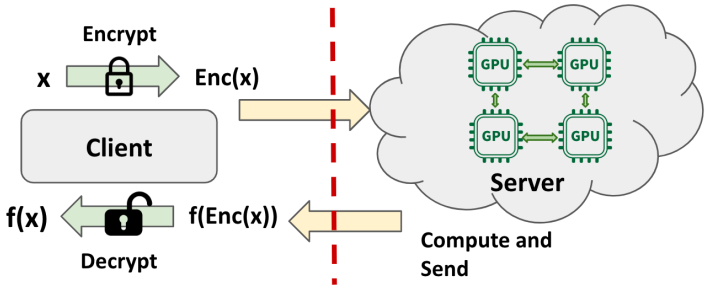

Figure 1.1: FHE allows for secure cloud computation

Recently, as interest in FHE grows, there has been a shift in focus to accelerate FHE workloads, especially through specialized hardware accelerators such as FPGAs, ASICs, and GPUs [10, 14, 15, 16, 17, 24]. Existing work clearly indicates that it is feasible to achieve large performance benefits for small ML models.

However, we observe a recent trend toward favoring large ML models, making it infeasible to apply the existing FHE acceleration models. Larger models increase the input size and also the number of ciphertexts required for encryption - raising the memory requirement. Also, they require more complex FHE operations such as bootstrapping that raise the compute requirement. Bootstrapping is an expensive ciphertext maintenance operation which can account for about 70-80% of the execution time of FHE programs.

The issue is that these models force larger monolithic chips and do not fit in the memory of GPUs. It was only recently that a scale-out framework for FHE, Cinnamon [13], was proposed, leveraging parallelism inherent in FHE workloads to efficiently split the requirements of large models across multiple chips. However, Cinnamon targets ASICs, which is expensive to produce, making widespread adoption infeasible. This points to the necessity to move to a scale-out design for GPUs in order for FHE to be applicable.

1.1 Thesis Contributions

In this thesis, we introduce Caramel, a multi-GPU backend for FHE that takes a step towards accelerating large models in a way that can be easily utilized in modern datacenters. Our work makes the following primary contributions:

- **Multi-GPU Design**: We introduce a general strategy for program and limb-level parallelism in FHE, and demonstrate its implementation across multiple NVIDIA GPUs using NCCL for high-bandwidth, low-latency communication.
- **GPU Overhead Reduction** We identify key overheads/bottlenecks in GPU FHE programs - namely, kernel launch overheads, inter-kernel memory traffic, and (I)NTT. We then present analyses of mitigation strategies, including batching kernel launches, constructing an execution graph for asynchronous execution, kernel fusion techniques, and various NTT configurations.
- **Other Exploration of Optimizations** We also present our progress in some unsuccessful optimization directions, including the use of OpenAI Triton and utilizing floating-point units for integer modular arithmetic.

Caramel achieves over $200\times$ speedup relative to a state-of-the-art CPU implementation and a $9.87\times$ improvement over our single-GPU baseline.

Chapter 2

Background

2.1 Fully Homomorphic Encryption

In this thesis, we focus on the CKKS encryption scheme [7], a FHE scheme for encrypted computing on real numbers. It is known for its high throughput and efficient error handling, compared to other FHE schemes. CKKS supports 3 homomorphic operations on encrypted values: addition, multiplication, and rotation. To encrypt, CKKS first batches multiple plaintext values into a vector, which is then encrypted into a single ciphertext - represented as a pair of polynomials. These polynomials are elements of a ring with dimension N with a large integer modulus.

Limbs/Residual Number System (RNS): Since the polynomials are taken with a large integer modulus (say q), the coefficients are also large, making modular arithmetic extremely inefficient. To resolve this issue, we decompose the modulus into the product of smaller prime moduli $Q = \{q_0, \cdots, q_{l-1}\}$ such that $q = q_0 \times \cdots \times q_{l-1}$. Then we apply the residual number system (RNS) [5] to represent each ciphertext polynomial as a tuple of smaller polynomials, which we call *limbs*. Figure 2.1 displays this construction.

NTT: The Number Theoretic Transform (NTT) is a specialized form of the Fast Fourier Transform (FFT) that operates under a finite integer modulo field. This operation performs a convolution and transforms polynomials between coefficient and evaluation domains, for the primary purpose of speeding up modular polynomial multiplication. The inverse NTT (INTT) reverses the transformation. The set of operations (NTT, INTT) will later appear to be a bottleneck in our GPU implementation.

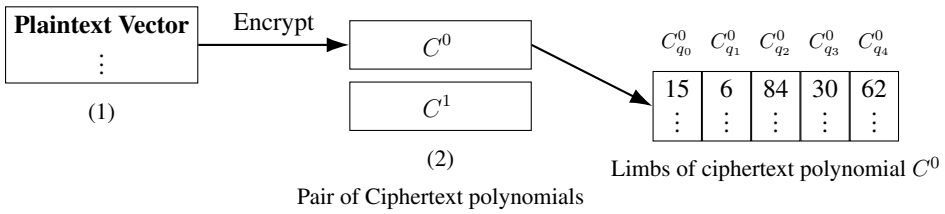

Figure 2.1: CKKS Plaintext and Ciphertexts with RNS Decomposition

Keyswitching: Keyswitching is a homomorphic operation that is critical to implementing homomorphic multiplication and rotation (2 of the operations supported by CKKS FHE). It takes a ciphertext that was encrypted using secret key k and converts it to an equivalent ciphertext encrypted using secret key k'. Keyswitching is computationally expensive and has been a major point of focus for previous acceleration works.

Bootstrapping: Each ciphertext has a finite multiplicative budget that is consumed by each multiplication operation performed on the ciphertext. This budget is a representation of the noise tolerance of the ciphertext (as ciphertext multiplications result in error terms). Once this budget is exhausted, computation cannot proceed until the budget is refreshed. Bootstrapping is a homomorphic procedure that refreshes this budget by raising the ciphertext to the highest multiplicative budget (or level) and then performing some additional computations. This additional processing consumes part of the total multiplicative budget as well. Figure 2.2 displays a ciphertext's multiplicative budget over time. Bootstrapping is extremely expensive, often taking 70-80% of execution time in FHE programs.

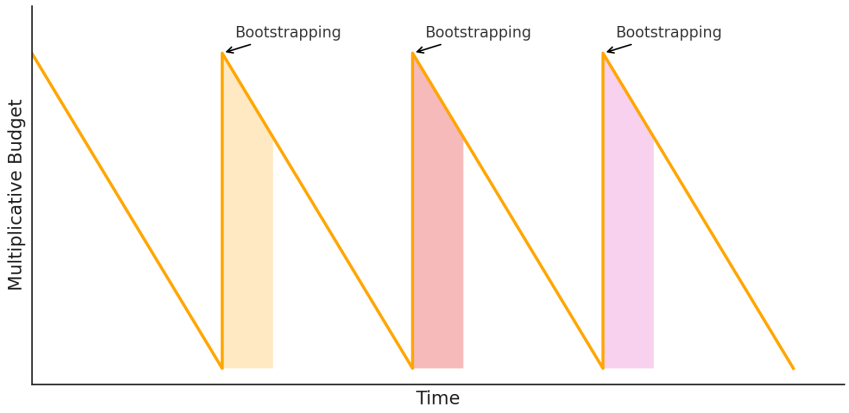

Figure 2.2: Ciphertext's Multiplicative Budget Over Time with Bootstrapping

2.2 Parallelism

To parallelize across multiple GPUs, we must understand the challenges and opportunities that come from larger workloads. First, larger workloads indicate larger input and activation values, meaning we must maintain more ciphertexts to store the encrypted data. Though more encrypted data imposes memory constraints, it also opens up opportunity for parallelism across multiple ciphertext - which we call *program-level parallelism*. Additionally, larger workloads perform more operations (eg. deeper programs), thus consuming more levels and requiring more bootstraps. Since these operations are on polynomials limbs that form a ciphertext, there are two main axes of parallelism - *coefficient-level* and *limb-level*. As the names indicate, coefficient-level parallelizes across the coefficients in a polynomial, where as limb-level parallelizes along coefficients across polynomials. Keyswitching has all-to-all dependencies across limbs, whereas NTT and automorphisms have all-to-all dependencies across coefficients.

2.3 Cinnamon

Cinnamon [13] is a framework for scale-out encrypted AI that introduces novel methods to exploit parallelism in FHE and targets the parallelism via a multiple ASIC system. We provide a description of key components of Cinnamon that we build upon.

2.3.1 Parallel Keyswitching Algorithm

Cinnamon introduced a novel parallel keyswitching algorithm to reduce the number of all-to-all communications required when employing limb-level parallelism. A sequential keyswitching algorithm proceeds by first partitioning the data limbs and temporary values necessary for computation (we call this the *extension basis*). We label these partitions as *digits*. Each digit then undergoes a mod up operation and polynomial multiplication to yield 2 evalkey product polynomials. These polynomials are aggregated across each digit and undergo a mod down operation.

Typically, a naive parallel keyswitch algorithm requires 3 broadcasts, one for the input limbs and extension basis, and two for each aggregated evalkey polynomial - which introduces a massive overhead. Cinnamon utilizes a novel reordering of instructions and partitioning of data to restrict the communication to a broadcast before the mod up operation and two aggregations after the mod down operation - greatly reducing the communication overhead.

2.3.2 Cinnamon ISA

Cinnamon also introduces an instruction set architecture (ISA) that operates on limbs (28-bit vectors of 64K elements). The limbs are placed into 'registers,' which operations use to refer to data. It supports computational instructions such as add, subtract, multiply, rotate, and (I)NTT, but also includes communication instructions for broadcasts (`dis,rcv`) and aggregation (`joi`) (as required by keyswitching). Figure 2.3 shows snippets of a FHE program compiled down to the Cinnamon ISA.

```
evg r8: k4821(0){F}
mul r9: r0[X], r8[X] | 0
rcv @ 131073:2 r10:
bci B1: [0,63], [1]
pl1 B1: r10[X] | 1
ntt r11: b1{1} | 63
...
joi @ 720946:2 r890: r723[X] | 50
joi @ 720945:2 : r285[X] | 49
...
```

Figure 2.3: Example of Cinnamon ISA

2.4 GPU Execution Model

Graphics processing units (GPUs) are highly parallel processors that are composed of streaming multiprocessors (SMs), which are analogous to cores on a CPU. Each SM is composed of SM sub-partitions that share a L1 cache, where each sub-partition contains a scheduler, register files, and numerous compute cores dedicated to specific types (eg. `int32`, `float32`, `tensor`). GPU programming revolves around the launching of a kernel function, which assigns the function to *thread* resources in a SIMD-model. Threads run the same instruction on different data, where data is assigned to each thread. Multiple threads form a thread-block, where multiple blocks are organized onto a grid. It is important to know that *blocks* (not individual threads) are scheduled onto the SMs, meaning the L1 cache cannot be shared by threads in different blocks. Figure 2.4 displays a high-level diagram of the GPU architecture, where blue boxes represent threads and green boxes represent a streaming multiprocessor, which contain a shared memory unit as shown in purple. Note for space, we omit from the diagram that one GPU has several (usually dozens) of SMs.

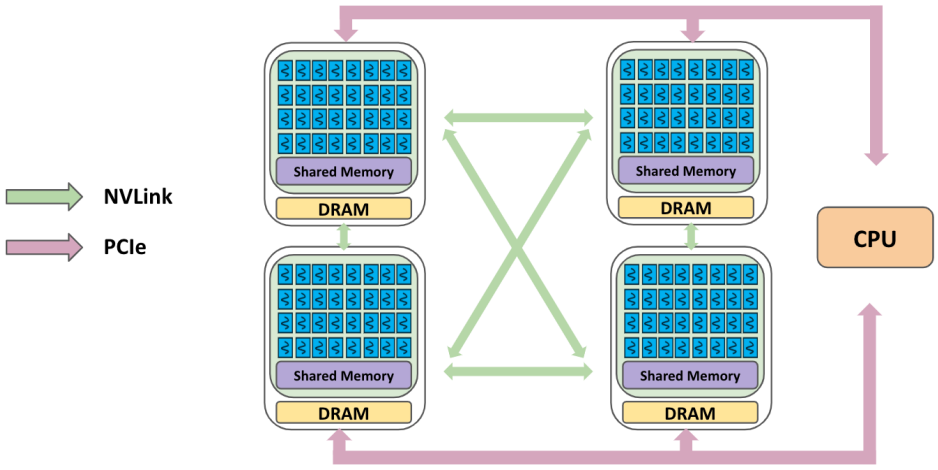

Figure 2.4: Multi-GPU System Design

GPUs also support a *stream* abstraction, which forms a queue of actions to be run on the GPU. By default, host code runs asynchronously from device code, but device code runs synchronously with respect to each other. By assigning independent kernels to separate streams, we can schedule them to run asynchronously. CUDA graphs is a library that uses this concept, allowing programmers to define an execution graph of kernels, which the library uses to run independent execution paths on separate streams. The other main contribution of CUDA graphs is that it batches kernel launch overheads.

Shifting our focus to a multiple GPU system, the GPUs communicate via NVLink [21], a wire-based communication link for NVIDIA GPUs. Figure 2.4 also displays a mesh topology of GPUs in a GPU cluster connected by NVLink (also with PCIe connections to the CPU). It exhibits higher bandwidth and lower latency than standard PCIe links, making multi-GPU designs feasible and efficient. We utilize the NVIDIA Collective Communication Library (NCCL) [20],

a lightweight GPU library that provides MPI-like collective primitives (all-reduce, broadcast, all-gather, ect...) to deliver near–wire-speed performance on multi-GPU systems. NCCL is aware of the interconnect topology and partitions the participating GPUs upon initialization into logical communication structures that maximize performance.

2.5 OpenAI Triton

OpenAI Triton [22] is a Python-embedded domain-specific language (DSL) that aims to relieve programmers from the complexity of GPU programming. Kernels are defined by a `@triton.jit` decorator with builtin primitives for vectorized loads/stores, arithmetic, and synchronization. Triton attempts to perform many optimizations in its compiler passes, such as memory coalescing, kernel fusion, and instruction scheduling, before it emits PTX or SASS kernels.

2.6 Prior Work

There has been an increasing effort to accelerate FHE using various types of hardware. Many earlier works targeted specialized hardware. CraterLake [24] explored an ASIC approach for unbounded-depth FHE programs, with fully-pipelined functional units for NTT, key-switching, multiplication, and rotation. It also introduced a compiler-driven data-placement that minimizes off-chip traffic for large (multi-MB) ciphertexts. CiFHER [17] identified concern with the construction of monolithic chips and proposed the use of limb-level parallelism across multiple FHE chiplets on the same package. Other notable accelerators include [3, 4, 15, 23].

Moving on to FHE on GPUs, 100x [14] identifies off-chip memory bandwidth, rather than raw compute, as the key GPU bottleneck for CKKS bootstrapping. They introduce stage reordering and amortized data transfers across thousands of threads to reduce the bootstrapping cost on GPUs. TensorFHE [10] was the first to leverage NVIDIA's Tensor Cores for FHE by recasting the NTT as a series of small matrix–matrix multiplications. Cheddar [16] is the current state of the art single-GPU FHE library, where it introduces optimizations on both low-level kernels and high-level operation scheduling. It introduces 32-bit word NTT kernels tailored to GPU thread-block layouts, fused base-conversion algorithms that eliminate extra synchronizations, and a scheduler.

Chapter 3

Caramel: Multi-GPU Backend for FHE

Caramel provides multi-GPU support for FHE that leverages a scale-out design targeting program and limb-level parallelism. It can act as a standalone FHE-GPU library and also as a GPU backend to the Cinnamon framework [13]. Caramel's main contributions are providing a foundation for multi-GPU communication in FHE programs, employing various techniques to lower overheads inherent in GPU programs, and exploring the use of float compute units for integer modular arithmetic.

3.1 Baseline Model

Caramel serves as a backend for Cinnamon, meaning we provide GPU implementations for all instructions defined by the ISA. First, we support modular arithmetic kernels (eg. add, sub, multiply) that are launched with a configuration of 256 blocks with 256 threads. This creates a one-to-one mapping of threads to data, recalling that limbs are of length 64K. It also optimally distributes work across thread-blocks (remember that each block is scheduled onto SMs).

Modular Multiplication: One instruction we focused on was modular multiplication. Integer division, which is used for modular reduction, is an expensive operation on GPUs. Instead, we use Barrett multiplication [6] which pre-computes a *Barrett ratio* representing $x = \lfloor \frac{b*2^k}{mod} \rfloor$ (fixed constant k), for product $a * b$. Then we find: $ab \ mod \ n = ab - \lfloor \frac{a*x}{2^k} \rfloor * mod$, where we avoid division by using arithmetic shifts.

Number Theoretic Transform: We also introduce optimal (I)NTT implementations, taking a radix-2, 8-point NTT approach. The point refers to the length of the local sub-transform, taking 8 input values. Radix refers to the size of the subproblem, where radix-2 indicates we split our problem in halves at each iteration. This configuration indicates we require 16 stages of butterfly iterations for an input size of 64K. We assign each thread an 8-point calculation, where three stages of "butterfly" operations occur, each stage pairing up elements and combining them with appropriate modular multipliers. The blue box in figure 3.1 displays the 8-point transform in a single thread. We then continue the transform across threads (in the same thread block) by clever indexing in shared memory. Recall that shared memory is a section of the L1 cache in

each streaming multiprocessor. Since all threads in a thread-block share this region of memory, we can speedup intra-block memory accesses. We launch the kernel with a block configuration such that the next 7 stages of transforms are contained within the same thread-block. The red box in figure 3.1 represents the intra-block, cross-thread transforms. For the remaining 6 stages, we require cross-block accesses and thus a synchronization, as we cannot ensure the completion of the sub-transforms in every block. This synchronization is achieved by exiting the kernel once all intra-block transforms have completed and launching a new kernel (which we denote NTT-Stage 2). Finally, we perform a clever permutation of data (via indexing) such that all remaining memory accesses are contained within one thread-block, and proceed to complete NTT.

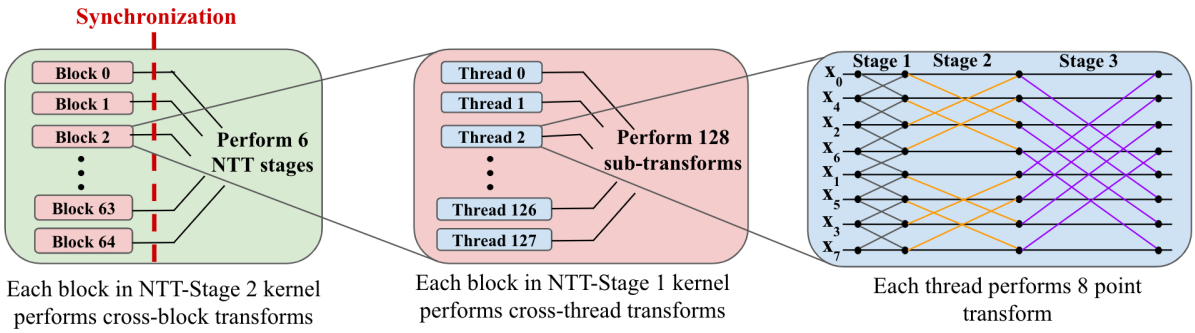

Figure 3.1: NTT Butterfly Iteration Tree

3.2 Multiple GPU Communication

FHE programs require all-to-all communications when performing keyswitches. The Cinnamon ISA embeds communication operations in the instruction streams; we explore optimal GPU communication strategies for these instruction. A baseline communication strategy was to utilize the CPU as a central resource manager. A GPU would share the data in a particular 'register' and other devices could read from the same location in CPU memory. Similarly, GPUs could send data to the CPU and offload the aggregation to the CPU. However, the host to device transfer overhead is massive, making this strategy infeasible. Instead, leveraging the NCCL interface, we introduce a ring-based communication for GPUs, calling the broadcast primitive for send and receive calls (used in sharing the input data) and the reduce primitive for aggregation calls.

3.3 Reducing Overhead

FHE programs are characterized by many small kernels, which accumulate large kernel launch overheads - we explore techniques to reduce this overhead. We first use CUDA graphs on a sequential stream to batch these launch overheads. We also iterate through the instructions and create a data dependency graph where nodes represent kernels. Edges are created between nodes if they have write-write, read-write, or write-read dependencies. In this way, we ensure the orig-

inal ordering as defined in the program. Using the CUDA graphs API, we can run independent branches of this execution graph asynchronously on different streams, reducing the overhead from lockstep execution.

3.4 Kernel Fusion

3.4.1 Fusion Techniques

Kernel fusion is a critical technique in GPU programming that merges several kernels operating on the same data into a single kernel. This reduces kernel launch overheads and inter-kernel memory traffic (eg. handling arguments in a single global memory load/store). The main constraint is that FHE prohibits certain operations from being fused; (I)NTT being prime examples. As previously mentioned, (I)NTT performs data accesses across blocks, requiring a split into 2 stages. This synchronization is accomplished by exiting the kernel and launching a second NTT kernel. The two stages of NTT cannot be fused, so we mark (I)NTT as a non-fusible kernel.

Keeping this restriction in mind, we perform a mix of maximal vertical and horizontal fusion. We create a kernel data dependency graph and start at the program entry node. We iterate down each branch in a depth-first-search manner and maximally fuse kernels until we reach an operation that we cannot fuse. We repeat this process across all branches until we have iterated through every node. This results in a smaller tree with a mix of fused kernels and non-fused kernels.

Since calls to non-fusible kernels are frequent, we also check for horizontal fusion opportunities. Looking at a level in the dependency tree, we see if there are independent NTT/INTTs that can be fused. Note that we cannot fuse the 2 stages of one NTT operation, but we can fuse the stage 1 kernels of multiple independent NTTs (same for stage 2 and INTT). This horizontal, breadth-first-search-like approach works because we are ensured that nodes in separate branches are independent. Figure 3.2 provides an example of our kernel techniques.

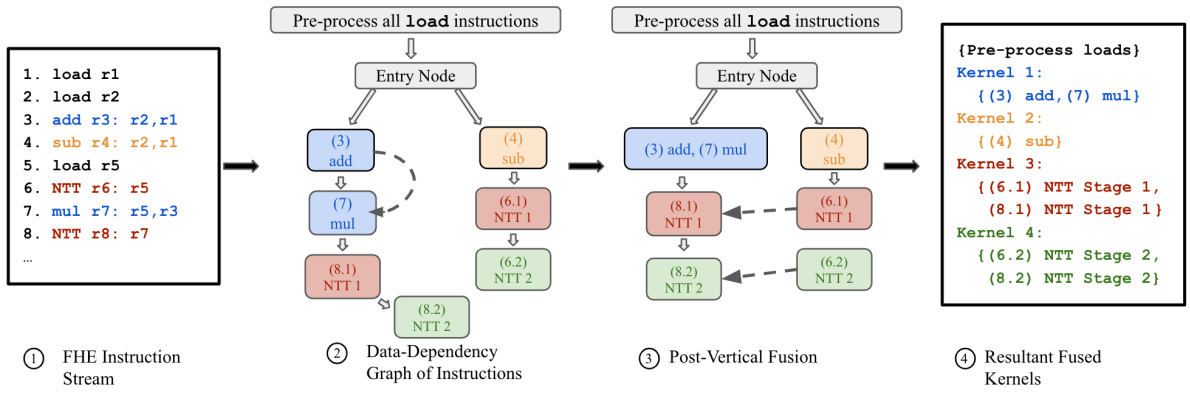

Figure 3.2: Example of Kernel Fusion

3.4.2 Fusion via Triton

OpenAI's Triton offers many GPU optimizations, including kernel fusion, under the hood, which we attempt to utilize. This requires a full remapping of the Cinnamon ISA to a Triton backend. The main difference from the CUDA implementation was the use of vectorized instructions, meaning we could no longer easily manipulate data accesses. This resulted in a shift in NTT design to follow a more standard, simplified butterfly iteration. It should be noted that Triton hides most GPU features from the programmer, limiting the flexibility in specific optimizations we could make (eg. allocating shared memory).

3.5 Optimizing GPU Utilization

We notice that some data values are used only once and others are reused multiple times. For example, the original data limb is only loaded and used once during NTT and the final results between the 2 NTT stages are not immediately reused. Thus we inline PTX instructions, providing cache-hints to the CUDA compiler. Using operations such as `ld.global.nc` (load with no caching) and `st.global.na` (store with no allocate), we prevent these one time accesses from evicting other data that exhibit good locality.

Lastly, we notice that modern GPUs are equipped with double the amount of float compute cores compared to integer compute cores. However, FHE operations are comprised only of integer modular arithmetic, indicating that theoretically, we miss out on half of the GPU's potential computation power. To target this opportunity, we introduce a double implementation of FHE - supporting double modular addition/subtraction and double modular multiplication (most other functions call these modular operations). Double modular addition is trivial, but efficiently implementing modular multiplication is a struggle. Currently, we utilize Barrett multiplication because integer division is expensive; however, floats support efficient division and are incompatible with arithmetic shifts, so we take a different approach.

First, we cast all inputs from integers to doubles upon entering the kernel (and re-cast to integers at before returning). We do not allow for casting during computation, as casting is an expensive operation. We perform $a * b * \frac{1}{mod}$ and add $2^{52} + 2^{51}$ in a single `fma` instruction. Referring to the IEEE standard for doubles [1], we know we have 51-bit integer precision, leading us to choose a constant that overflows when added - forcing the value to round to an integer. Note the native rounding instruction for doubles is expensive. We then subtract the constant to revert to a rounded, approximate quotient q. We then calculate error terms from this approximate quotient, or the overflow l from $mod * q$ and use this to calculate the final result $a * b - mod * q - l$.

Chapter 4

Methodology

To test the performance of Caramel, we created a large FHE program characterized by requiring bootstraps. Namely, we take a ciphertext at a ring dimension of $N = 64K$ at level 2, raise its level (budget count) to 51, and consume 36 levels to require bootstrapping - leaving 13 levels for the application. The program is implemented at 128-bit security and uses 28-bit primes for RNS decomposition. It should be noted that a CPU implementation of this program takes 33 seconds (33,000 ms).

Before proceeding, we note that this bootstrap workload is representative of the performance of encrypted inference models. Prior works [13, 24] have shown that 80% of execution time in encrypted inference is spent on bootstrapping. The pipeline of encrypted inference models is shown in figure 4.1. Essentially, if we make bootstrapping efficient, we can make encrypted inference efficient as well.

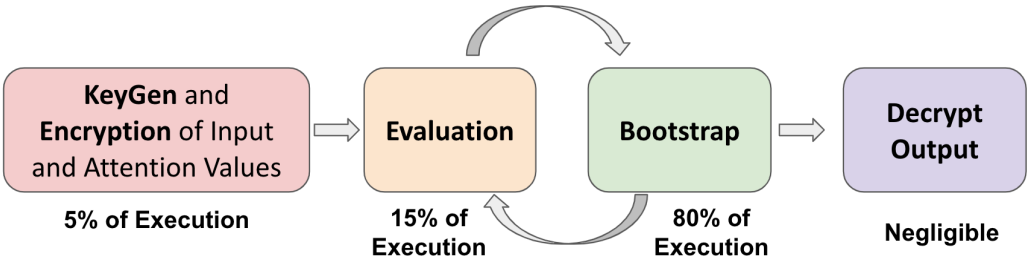

Figure 4.1: Pipeline of Encrypted Inference

Though our ultimate goal was to test on real-world models such as Resnet [12], HELR [18], BERT [9], and Llama [26], we unfortunately did not have time to structure these models into FHE schemes (eg. bootstrap placement) and fit them into the memory of large GPU systems. We also wished to run on larger GPU clusters and on various GPU models, but unfortunately, we did not have access to these hardware resources.

4.1 GPU Configuration

Table 4.1 shows the configuration of the GPU used to run Caramel on our bootstrap program. For our two GPU configuration, the two NVIDIA A30 GPUs are connected by a NVLink bridge with a bandwidth of 200 GB/s.

GPU	`int32` **Peak Throughput**	**DRAM (capacity, bandwidth)**
A30	10.3 TOPS	24 GB, 933 GB/s

Table 4.1: Configuration of GPU used (compute throughput and memory bandwidth)

Chapter 5

Results

5.1 Single-GPU Model

We first examine the effectiveness of our GPU optimization techniques. Figure 5.1 compares the execution time of the bootstrap program on the A30 with incremental designs.

Our baseline design has an execution time of 1450 ms, displaying a $22.8\times$ speedup from state of the art CPU implementations. However, when profiling the baseline design (using Nsight Systems), we observe 'gaps' (as in ②) between the execution of kernels as displayed in figure 5.2, where we desire execution patterns as in section ①. These spaces in the profiler represent various overheads that occur, kernel launch overheads being the main contributor. Furthermore, we found that we were spending approximately 60% of execution on overhead.

To resolve these gaps, we employ a sequential stream CUDA graphs design. By batching the overheads (ie. gaps), we achieve a $1.58\times$ speedup from our baseline implementation. Next, by extending CUDA graphs to an explicit execution graph design, we obtain an additional $2.3\times$ speedup, reaching an execution time of 400 ms. This shows the success in our initial optimizations to mitigate these overheads.

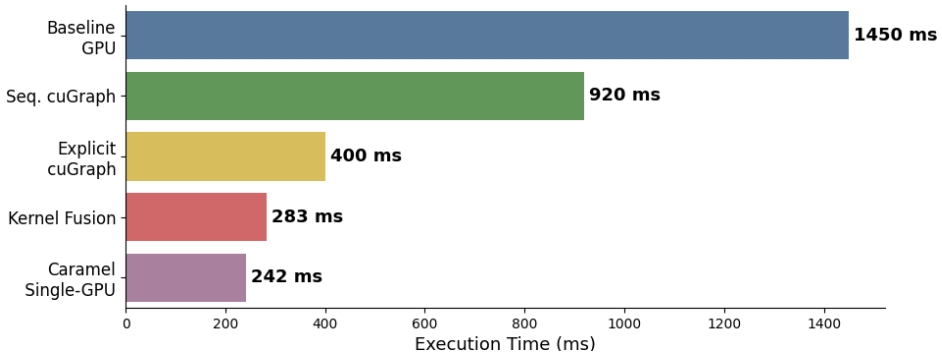

Figure 5.1: Execution Time across various FHE GPU Implementations

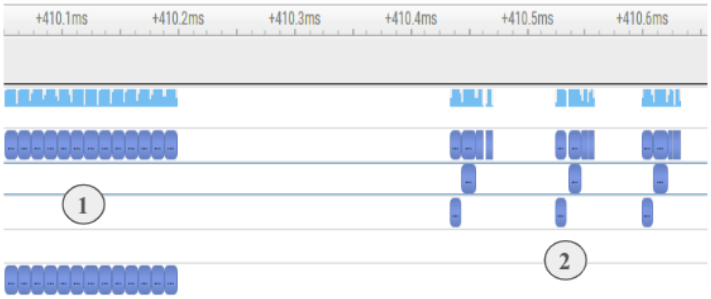

Figure 5.2: Profiler Results on Baseline

5.1.1 Kernel Fusion Analysis

Our attempt to utilize the kernel fusion techniques in Triton was unsuccessful. When running a smaller program than our bootstrap benchmark, we found that Triton ran approximately between $1.2 - 1.4\times$ slower than our baseline. This was initially a surprise, as our motivation for using Triton was to plug into its optimization layers. However, we realized we were targeting Triton's fusion techniques on an instruction layer that was too low. Triton targeted fusion of instructions within the same Triton kernel definition - not across kernels. The construction of Triton kernels on the Cinnamon ISA prohibited our desired fusion - perhaps if we structured the Triton backend for a higher Cinnamon intermediate representation (IR), we may have yielded better results.

This led to the development and analysis of our own kernel fusion techniques. Referring back to figure 5.1, we see that the addition of kernel fusion techniques results in an additional $1.43\times$ speedup. This result makes sense as CUDA graphs helps in batching and pipelining various overheads, not removing them. The result indicates that kernel fusion was effective in reducing the kernel launch and memory load/store overheads, as intended.

5.1.2 NTT Analysis

After targeting these various overheads, we looked to analyze the state of our program. FHE programs contain many (I)NTT instructions, where NTT is a relatively expensive operation. Through use of profiling tools (Nsight Compute), we find the results in figure 5.3. The diagonal lines indicate the bandwidth across different memory hierarchies and the top horizontal line represents the compute limit. The closer we are to the diagonal line, we are memory bound, and the closer we are to the horizontal line, we are compute bound. The figure demonstrates that we fully utilize the bandwidth along DRAM, but not along the L1 cache. The figure also shows that we have more compute that we can utilize.

To better utilize these resources, we analyzed different NTT configurations - namely a radix-4 and radix-8 NTT. Recall that the radix number defines the partitioning of elements per iteration (eg. radix-2 halves the problem size at each iteration). Analysis on FFT [27] setups revealed that radix-4 and radix-8 implementations lowers the arithmetic intensity but results in better memory

traffic patterns. However, upon implementing these configuration, we found that it was hard to apply the same optimal use of shared memory for higher radixes, which resulted in a significant dip in NTT execution times - leading us to utilize a radix-2 implementation.

Lastly, the lower utilization of L1 caches motivated the need for cache hints. This optimization gave us marginal benefits, providing a $1.2\times$ improvement from our kernel fusion design. The benefits are reflected slightly when repeating the roofline analysis, where the blue dots move slightly to the left (we omit the updated graph as the difference is hard to see).

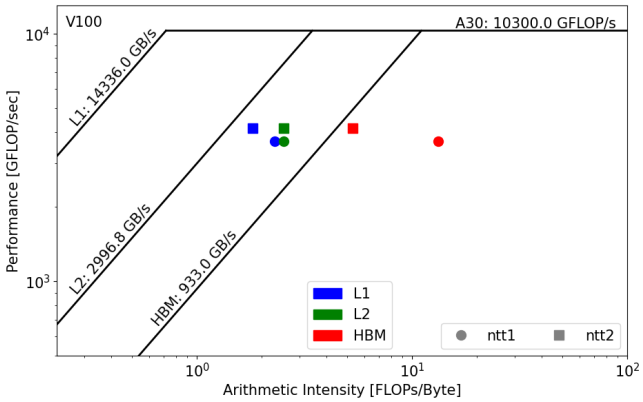

Figure 5.3: NTT Roofline Analysis

5.1.3 Floating Point Analysis

Unfortunately, our introduction of float techniques was not initially successful. Casting between types utilizes a limited compute resource, making it a low throughput instruction. Since we cast at the start and end of every kernel and we have many kernels, this casting instruction introduces a bottleneck, resulting in a $1.5\times$ slowdown.

5.2 Multi-GPU Model

Now we observe the scaling behavior of Caramel across multiple GPUs. Note that our analysis is limited due to our resource constraints. Observing figure 5.4, we notice near linear speedup from 1 to 2 GPUs, where we had a $1.64\times$ and $2.10\times$ speedup respectively across two critical implementation stages. Note that the explicit CUDA graph implementation shows a higher speedup because at this point in our design, we were memory bound - thus gaining more benefits from splitting the work across 2 devices. On the other hand, the current Caramel design has pushed to be more compute intensive, thus seeing less benefits from the added device.

These results indicate that our parallelism strategy seem to be effective on a multi-GPU system, though we cannot know for certain unless we scale on larger programs and more devices.

However, looking ahead, we note that we must be considerate of interconnection bandwidth and monitor if the cross-device transfers bottleneck the program (for large memory intensive programs).

Finally, Caramel achieves a $9.87\times$ speedup on 2 GPUs over our single-GPU baseline.

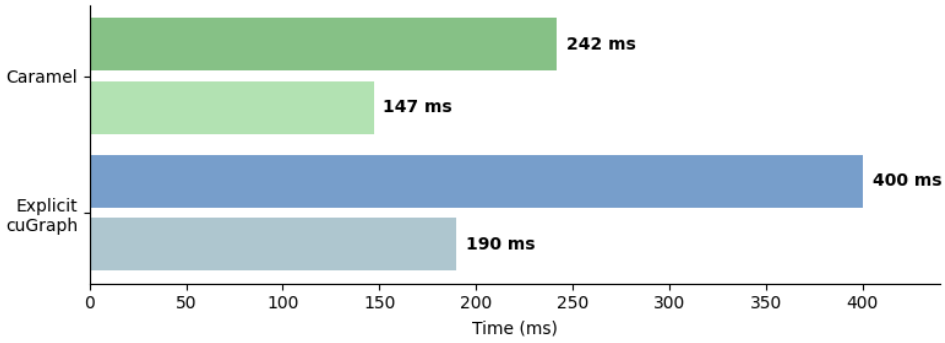

Figure 5.4: Caramel Scaling Across Multiple GPUs

5.3 Comparison with Prior Works

Cinnamon [13] utilizes an equivalent parallelism strategy as Caramel, but targets ASICs. Due to the massive compute power of specialized hardware, Cinnamon does not display good scaling for small programs (including our bootstrap program). However, when comparing the general scaling behavior of Cinnamon, we exhibit similar behavior of showing good speedup from one to two chips. Assuming a similar scaling trend as Cinnamon, we can expect good speedup that tapers off when adding more GPUs, due to the computational intensity of the program being overtaken by the extra compute from more devices.

Chapter 6

Conclusion

In this thesis, we presented Caramel, a novel multi-GPU acceleration backend for fully homomorphic encryption (FHE) that tackles the key performance challenges of FHE on GPUs. We present parallelism strategies for FHE workloads that are compatible with multiple GPUs and display a GPU implementation of FHE that targets the various overheads inherent in GPU programs. Our current work entails optimizations of data-dependency-based scheduling, kernel fusion, memory management, and use of float compute cores for integer operations. Caramel achieves over $200\times$ speedup relative to a state-of-the-art CPU implementation and a $9.87\times$ improvement over our single-GPU baseline.

6.1 Future Works

The evaluation of Caramel was limited by the availability of GPUs. Our immediate next step is to test Caramel's performance on various GPUs with varying configurations, and observe its scalability on clusters of 4 or more GPUs. Another immediate next step is to test Caramel on real world models, including smaller models such as Resnet [12] and larger ones such as BERT [9] and Llama [26]. It should be noted that testing on large models will require us to explore optimal memory mapping strategies, in order to fit the large input/attention values into the memory of the GPUs in a cluster.

Looking further ahead, we are looking to support an end-to-end framework for encrypted AI with support for increasingly larger models. Additionally, we look to make architectural extensions to the GPU to optimize for FHE-specific computation and memory needs - hopefully shaping the development of future GPU architectures.

Bibliography

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229. 3.5

[2] Heaan. Online: `https://github.com/snucrypto/HEAAN`, November 2023. 1

[3] 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, 2023. 2.6

[4] Rashmi Agrawal, Anantha Chandrakasan, and Ajay Joshi. Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping. 2.6

[5] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. Cryptology ePrint Archive, Paper 2016/510, 2016. URL `https://eprint.iacr.org/2016/510`. 2.1

[6] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. ISBN 978-3-540-47721-1. 3.1

[7] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017. 2.1

[8] Federal Trade Commision. Equifax data breach settlement. URL `https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement`. `https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement`. 1

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. 4, 6.1

[10] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–934, 2023. doi: 10.1109/HPCA56546.2023.10071017. 1, 2.6

[11] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585062. doi: 10.1145/1536414.1536440. URL https://doi.org/10.1145/1536414.1536440. 1

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. 4, 6.1

[13] Siddharth Jayashankar, Edward Chen, Tom Tang, Wenting Zheng, and Dimitrios Skarlatos. Cinnamon: A framework for scale out encrypted AI. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25. Association for Computing Machinery, April 2025. 1, 2.3, 3, 4, 5.3

[14] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. Cryptology ePrint Archive, Paper 2021/508, 2021. URL https://eprint.iacr.org/2021/508. https://eprint.iacr.org/2021/508. 1, 2.6

[15] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1237–1254, 2022. doi: 10.1109/MICRO56248.2022.00086. 1, 2.6

[16] Jongmin Kim, Wonseok Choi, and Jung Ho Ahn. Cheddar: A swift fully homomorphic encryption library for cuda gpus, 2024. URL https://arxiv.org/abs/2407.13055. 1, 2.6

[17] Sangpyo Kim, Jongmin Kim, Jaeyoung Choi, and Jung Ho Ahn. Cifher: A chiplet-based fhe accelerator with a resizable structure, 2024. 1, 2.6

[18] Han Kyoohyung, Seungwan Hong, Jung Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:9466–9471, 07 2019. doi: 10.1609/aaai.v33i01.33019466. 4

[19] Natasha Lomas. Italy orders chatgpt blocked citing data protection concerns. URL https://techcrunch.com/2023/03/31/chatgpt-blocked-italy/. https://techcrunch.com/2023/03/31/chatgpt-blocked-italy/. 1

[20] NVIDIA. Nccl, . URL https://developer.nvidia.com/nccl. https://developer.nvidia.com/nccl. 2.4

[21] NVIDIA. Nvlink, . URL https://www.nvidia.com/en-us/data-center/nvlink/. https://www.nvidia.com/en-us/data-center/nvlink/. 2.4

[22] OpenAI. Triton. URL https://openai.com/index/triton/. https://openai.com/index/triton/. 2.5

[23] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S.

Lee, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference, 2020. 2.6

[24] 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. ISBN 9781450386104. doi: 10.1145/3470496.3527393. URL https://doi.org/10.1145/3470496.3527393. 1, 2.6, 4

[25] SEAL. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA. 1

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971. 4, 6.1

[27] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992. doi: 10.1137/1.9781611970999. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611970999. 5.1.2

[28] Wired. The snowflake attack may be turning into one of the largest data breaches ever. URL https://www.wired.com/story/snowflake-breach-advanced-auto-parts-lendingtree/. https://www.wired.com/story/snowflake-breach-advanced-auto-parts-lendingtree/. 1