Parallel Polynomial Evaluation Algorithms for Leveled Fully Homomorphic Encryption

Vele Tosevski
University of Toronto
vele.tosevski@mail.utoronto.ca

Abstract—**Computation in a Leveled Fully Homomorphic Encryption (LFHE) context is more inefficient than in a plaintext (PT) context. Latencies grow as operations on ciphertexts (CTs) grow, limiting the efficiency of larger computations like polynomial evaluations (PEs). In addition, the amount of successive CT multiplications is cryptographically bounded, limiting the maximum degree of polynomials that can be evaluated. As such, we explore three different polynomial evaluation algorithms for Leveled Fully Homomorphic Encryption (LFHE) with the following properties: algorithms are (1) parallelizable in a multi-core environment to hide LFHE operation latency, and (2) strive to minimize CT-CT multiplicative depth to increase the amount of operations that can be applied afterwards and decrease the latency of operations. We show that Estrin's algorithm is the best with high computational latencies, achieving a near linear speedup in execution time as the degree of polynomial grows. We show that Dorn's algorithm becomes optimal as latencies approach plaintext levels.**

I. Introduction

With cloud services expanding and the Internet of Things growing exponentially, data security has never been more important. Several times in a day, consumers transmit data to cloud services and receive data back. To maintain data confidentiality, data is encrypted at its source and decrypted at its destination. This ensures that an adversary controlling the internet cannot decrypt messages in transport. At its destination, data must be decrypted in order to be used in computation. This leaves a vulnerability open for adversaries in control of the destination. If data can be extracted during computation in plaintext (PT) or non-encrypted form, its security is broken. To close this vulnerability, encryption schemes need to facilitate computations on data while it is encrypted.

A. *Fully Homomorphic Encryption*

The first such scheme was introduced by Gentry in his doctoral thesis [1] as recently as 2009. Fully Homomorphic Encryption (FHE) allows an arbitrary amount of additions and multiplications on encrypted data. While it was an extraordinary breakthrough, Gentry's scheme was proven to be very computationally intensive and thus impractical for common use [2]. This was a result of the *bootstrapping* procedure necessary to allow an arbitrary amount of computations. Continuous FHE operations add noise to ciphertexts (CTs). This noise is necessary for encryption but limits the amount of computations possible on a CT. When the noise reaches a large enough value, CTs cannot be decrypted properly. Bootstrapping generates a "fresh" CT from an existing CT with a renewed noise budget. By induction, this allows an arbitrary amount of computations on CTs.

To reduce the computational complexity of FHE, researchers created the first Leveled Fully Homomorphic Encryption (LFHE) scheme, BGV [3]. BGV removed the bootstrapping procedure but introduced a new limitation on FHE circuits: multiplicative depth (depth from hereafter). Under the BGV scheme, engineers are allowed to perform an arbitrary amount of additions but only an apriori set amount of multiplications. Because BGV made FHE computations more practical, most schemes introduced after have been implemented as LFHE schemes (most notably, B/FV [4], CKKS [5]).

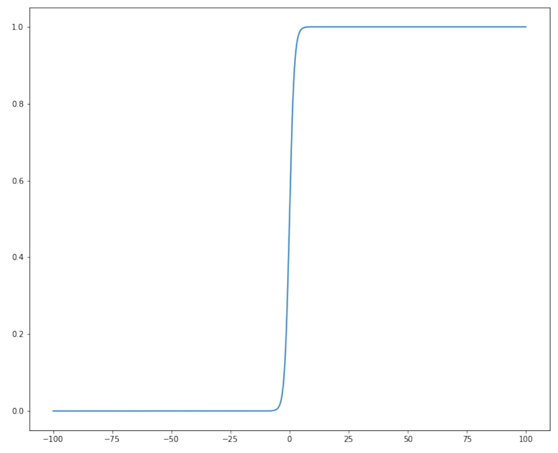
A limited depth makes it necessary for engineers to both practically and theoretically optimize their solutions. One of the most important and basic mathematical circuits are polynomials. Polynomials contain chains of additions and multiplications. As such, evaluating polynomials in LFHE requires multiple LFHE addition and multiplication operations, the total depending on the degree of the polynomial. The depth therefore determines the number of polynomials that can be evaluated: the number of CT-CT multiplications in a single polynomial must not exceed the depth.

B. *Privacy Preserving Machine Learning*

One of the most prominent applications of FHE is Privacy-Preserving Machine Learning (PPML). PPML allows third-parties to perform training or inference on encrypted data, providing services to users in need of their data to remain private. Ever since CryptoNets [6], the first Neural Network (NN) adaptation of FHE, multiple papers have investigated a variety ML models ([7], [8], [9], [10], [11]). ML also consists of many addition and multiplication operations. However, many operations do not include successive CT-CT multiplications. Matrix-vector multiplications (or fully connected layers) have a depth of at most 1 since there is one multiplication. Non-linear layers, where an activation function is applied, can potentially have multiple CT-CT multiplications.

1) Activation Functions: A non-linear layer in ML consists of an activation function being applied to a vector. An activation function is generally a non-linear function. Unfortunately, FHE cannot evaluate anything other than multiplications and additions leaving core ML activation functions like Sigmoid and Rectified Linear Unit (ReLU) non-implementable.

To circumvent this problem, researchers have used polynomial activation functions (such as a simple square function [6]). While they have been found to be problematic by researchers when considering the expressiveness of a NN ([6], [12]), they still can be applied to achieve good results. Some papers also use polynomial approximations to activation functions like the Sigmoid [7] and ReLU [13]. Taylor series and Chebyshev series polynomials [14] are usually used as good, bounded, approximations to activation functions. Figure 1 illustrates how an order-15 Chebyshev approximation to the sigmoid function compares to the actual sigmoid function.


(a) Real Function

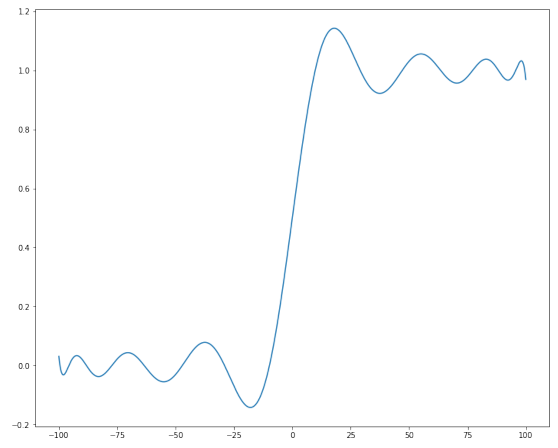

(b) Order-15 Chebyshev Approximation

Fig. 1: Sigmoid Function

Visually, the functions are very similar. Numerically, they are also very similar [1]. These figures show that as long as a NN is designed to bound inputs to activation layers, polynomial function approximations can be used.

This work focuses on evaluating three polynomial evaluation algorithms with respect to execution time and largest chain of multiplications on a CT. Section II introduces three polynomial evaluation algorithms that have already been proposed that we evaluate in this work. We discuss the algorithms, their relevance to our work, and evaluate theoretically how they might affect polynomial evaluation in FHE. Section III discusses our methods for evaluating the three polynomial evaluation algorithms analyzed in section II. Section IV showcases our results and provides our interpretation and how our results add to the field. We finish with concluding remarks in V and some proposed future directions of research.

[1]We leave an empirical analysis of this for future work.

II. Related Work: Polynomial Evaluation Algorithms

Athough LFHE schemes without bootstrapping have been introduced to make FHE more practical, regular addition and multiplication operations in FHE still take many more compute cycles to execute than regular PT operations. We show this by evaluating addition and multiplication operations with SEAL, Microsoft's **S**imple **E**ncrypted **A**rithmetic **L**ibrary [15]. Native to C++, Microsoft SEAL is a leading FHE open-source library. Using SEAL's highest setting (which allows the largest depth) and the CKKS scheme, we found that the average CT-CT multiplication executes for 37,793μs and the average CT-CT addition executes for 2376μs. However, CT-CT multiplications must be immediately followed by a relinearization operation to reduce the size of CTs as a best practice [15]. We found that the average relinearization operation executes for 328,987μs, a substantial amount. Therefore, a multiplication effectively executes for 366,780μs, the combination of both. In comparison, the average PT-PT addition and multiplication execute for 0.023μs and 0.026μs respectively (see section III for our method of evaluation). Due to the large latency of regular FHE operations, naïvely evaluating polynomials in FHE would drastically slow down PPML with polynomial-approximated activation functions. As such, algorithms for evaluating polynomials are an important area of research for PPML design.

For the following subsections, we refer to a polynomial as:

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0 \tag{1}$$

A. Naïve Algorithm

A naïve algorithm for evaluating polynomials proceeds as follows: (1) calculate all powers of a polynomial by multiplying each x^n serially (ex. $x^3 = x * x * x$), (2) multiply each coefficient a_i by its corresponding x^i and add all together for the final result. This algorithm is purely sequential and does not use any parallelism. It also is not data efficient meaning it does not use previously computed values to its benefit. As a result, we discard this algorithm and investigate others.

B. Horner's Algorithm

Horner's algorithm [16] is an O(n) algorithm for computing polynomials. It is sequential and does not use any parallelism but is very data efficient. It is an "online" algorithm where previous computed results are built upon to achieve a final result. The algorithm performs as follows:

$$f(x) = (...((a_n x + a_{n-1})x + a_{n-2})x... + a_1)x + a_0 \tag{2}$$

(1) taking the highest order coefficient, multiply it by x and add the next coefficient in decreasing order, (2) repeat by multiplying the result with x and adding the next coefficient in decreasing order until a_0. If the coefficients and variables are both CTs, we see that there are each O(n) multiplications and additions in a row. This forces the depth of an FHE circuit to be at least n, the degree of the evaluated polynomial. Horner's algorithm is therefore not the best choice for LFHE evaluations but can serve as a good baseline for performance in terms of execution time. We expect Horner's algorithm to have a linear execution time with respect to the degree of polynomials being evaluated.

C. Dorn's Algorithm

Dorn's algorithm [17], also referred to as kth-order Horner's algorithm, is essentially a parallel version of Horner's algorithm. Dorn proposes dividing a polynomial into $k \in [2, \lfloor \frac{n}{2} \rfloor]$ smaller polynomials and using Horner's method to evaluate each sub-polynomial on k different processors or threads, as shown in [18]:

$$f(x) = x^{k-1}p_{k-1}(x^k) + ... + xp_1(x^k) + p_0(x^k) \quad (3)$$

where

$$p_i(x^k) = a_{i+\lfloor n/k \rfloor k}(x^k)^{\lfloor n/k \rfloor} + ... + a_{i+2k}x^{2k} + a_{i+k}x^k + a_i \quad (4)$$

for $i \in [0, k-1]$.
The algorithm proceeds in three steps:

1) Compute x^k using any method [2],
2) Evaluate k sub-polynomials (Eq. 4) on k different processors using Horner's algorithm,
3) Combine them as in Eq. 3 and compute the final result using Horner's algorithm.

Munro and Paterson [20] find that Dorn's algorithm, with k processors, runs in time at least $2n/k + 2\log k$. As such, with large n and $k = \lfloor n/2 \rfloor$, we expect Dorn's algorithm to outperform Horner's. A short analysis shows us that $\log k$ multiplications are initially performed on x (using a power tree [19]). Subsequently, in step 2, the largest exponent each processor raises x^k by is $\lfloor \frac{n}{\lfloor n/2 \rfloor} \rfloor = 2$ which can be derived from (4). A coefficient is also multiplied to this value making the maximum multiplication chain equal to 2 for each processor in step 2. Step 3 multiplies the largest value from step 2 to an already computed x^{k-1} which is computed in $\log(k-1)$ successive multiplications. Since $\log(k) + 2 > \log(k-1)$, we take the largest value form step 2 as the largest depth and add 1 since we are multiplying one last time. Therefore, the largest multiplication chain is $\log(k+3) < n$. This is better than Horner's in terms of depth and execution time making it a good candidate for testing LFHE polynomial evaluation [3].

[2]Could be a data efficient method (ex. $x^4 = x * x * x^2$) with less multiplications in a row as summarized by Knuth, ch.4.6.3 in [19], or the naïve sequential way (ex. $x^k = x * x * x * ... * x$).

[3]We leave a formal proof for future work.

D. Estrin's Algorithm

Estrin's algorithm [21] is also a parallel, data efficient algorithm. Based on the number of coefficients $c = n + 1$, Estrin also proposes to divide a polynomial into smaller sub-polynomials but rather than only once like in Dorn's algorithm, a $\log c$ amount of times.
The algorithm proceeds as follows:

1) Divide coefficient array into $\lfloor c/2 \rfloor$ groups of 2 $(\{a_0, a_1\}, \{a_2, a_3\}, ..., \{a_{\lfloor c/2 \rfloor -2}, a_{\lfloor c/2 \rfloor -1}\})$.
2) Allocate each group to $c/2$ processors and calculate $a_i + a_{i+1}x$. Also allocate x to one processor at the same time that updates $x = x * x$ (to be clear, $c/2 + 1$ processors execute at the same time in this step).
3) The $c/2$ processors return and fill a new array of $c/2$ members. The last processor returns the new value of x. Calculate the new value of $c = c/2/2 = c/4$. Repeat steps 1 and 2 until the coefficient array shrinks to only one member. Return the final member as the result.

Munro and Paterson prove that Estrin's algorithm runs in $2 \log n$ time, a big improvement to the previous two algorithms. Therefore, for $n >> 1$, we expect this algorithm to perform superior to any other algorithm discussed previously. A short analysis of Estrin's algorithm shows that the longest chain of multiplications is $2 \log n - 1$. We see this by observing the following: (1) in the second step, a CT coefficient is multiplied by a CT variable x, and (2), also in the second step, a CT variable x is multiplied by itself in preparation for the next repeat iteration. This shows that 2 successive multiplications to the same CT are performed at each repeat iteration of step 2. We know step 2 is repeated $\log n$ times. Discarding the last multiplication of x with itself (since it is unnecessary), we get a maximum of $2 * logn - 1$ successive multiplications to the same CT. This means our LFHE depth must be at least $2 \log n - 1$ for Estrin's algorithm to evaluate a polynomial in FHE correctly [4].

To the best of our knowledge, no other work has formally analyzed how to apply these three algorithms to FHE. PPML papers usually invest time in talking about the polynomial approximation they use as their activation function and quickly glance over how they evaluate their function in FHE in their methods section. While we do not provide formal proofs for the analyses above, we offer a good idea as to what types of algorithms can be applied and which of the three we evaluate are the best. The three algorithms we have chosen to evaluate are by no means the only ones proposed. They are, however, very popular and most other algorithms are built from them (ex. [18]). In the remaining sections, we empirically show which of the three algorithms mentioned is best for FHE evaluation of polynomials.

III. Methods

Most ML libraries (such as PyTorch [22] and TensorFlow [23]) make use of Graphics Processing Units (GPUs) to accelerate ML operations. We follow in the same direction

[4]Again, we leave a formal proof for future work.

and evaluate our three polynomial evaluation algorithms running on a GPU. To do this, we wrote all three algorithms in the native CUDA C++ language [24] for Nvidia GPUs in plaintext first, hoping to integrate Microsoft SEAL [15] later. Unfortunately, Microsoft SEAL as it is, without any modifications to the source code, did not compile with a regular CUDA program when just including the SEAL header file. After many weeks of debugging and trying to compile SEAL with nvcc, CUDA's native compiler, we did not make any progress. After speaking with the developers of SEAL, it was mentioned that it is not possible to migrate SEAL to CUDA since the whole library would need to be rewritten to include CUDA-specific C++ extensions. They estimated that such an endeavour would take anywhere between 2 weeks and 2 months of work, something we did not have at our disposal. Unfortunately, there are also no plans from the SEAL team to write a CUDA compatible version of the library. Casually glancing over the open and closed issues on SEAL's github page will prove that CUDA has been addressed but that no development has been done. There are other FHE libraries that use the CUDA language as their native language (such as cuHE [25], cuFHE [26], and NuFHE [27]). Unfortunately, time did not permit for us to switch directions and use a different library.

To mitigate this and still produce meaningful results, we measure the average execution time of a CT-CT multiplication and CT-CT addition on a CPU using SEAL and **speculatively add the measured latency to each PT multiplication and addition**. This means that every addition and multiplication in the C++ code written for CUDA for each algorithm has extra latency introduced to mimic the latency that we would experience with every operation when using SEAL.

We use SEAL v3.6.4 and build SEAL using no optimizations other than the regular C++ compiler -O2 optimizations. This is because we want to make latency evaluation on a CPU measure as close as possible to that on a GPU. As such, we do not use any extra build options such as Intel intrinsics and Intel HEXL [28] since the associated architectures do not exist on GPUs. We only enable the option to build SEAL with C++17 since our algorithms are written in C++17.

The tests were conducted with the setup described in table I. Using SEAL's built-in performance evaluation code and running it on the CPU, we extracted the CT evaluation metrics in table II.

To understand how well PT-PT operations run on the GPU, we timed individual operations and averaged them over 1000 trials. Table II shows PT-PT operations and how they are several orders of magnitude faster than CT-CT operations with SEAL. As mentioned in section I-B1, we added both the CT-CT multiplication and CT relinearization execution times to create an overall CT-CT multiplication execution time to follow best practice.

We use these execution times to introduce latency per matching operation in the C++ algorithms implementation. We simply add a timer that waits for the latency to finish to continue the algorithm. Where there are additions we add a round of addition latency and where there are multiplications, we add a round of multiplication latency, equal to the bold values in table II.

Component	Value
System	MSI GE62-2QF Apache Pro
CPU	Intel(R) Core(TM) i7-5700HQ CPU @ 2.70GHz, 2701 Mhz, 4 Core(s), 8 Logical Processor(s)
System Type	x64-based PC
Installed Physical Memory (RAM)	16.0 GB
Available Physical Memory	7.05 GB
OS Name	Microsoft Windows 10 Pro
Version	10.0.19042 Build 19042
GPU	NVIDIA GeForce GTX 970M
GPU Architecture	Maxwell GM204
Graphics Memory	3.0 GB GDDR5
Visual Studio Platform Toolset	Visual Studio 2019 (v142)
Windows SDK Version	10.0.18362.0
C++ Language Standard	ISO C++17 Standard (/std:c++17)
SEAL Version	3.6.4, Release
SEAL FHE Scheme	CKKS
SEAL Poly Modulus Degree	32,768
SEAL Security Level	128 bit
SEAL Coefficient Modulus Count	16
SEAL Coefficient Modulus Total Bit Count	881
CUDA Driver Version	GeForce Game Ready Driver v466.11, 04/14/2021
CUDA Toolkit Version	11.3.58

TABLE I: Testing Environment

Operation	**Average Evaluation Time** (μs)
PT-PT Multiplication	0.026
PT-PT Addition	0.023
CT-CT Multiplication	37,793
CT-CT Addition	**2376**
CT Relinearization	328,987
CT-CT Multiplication + CT Relinearization	**366,780**

TABLE II: Average Evaluation Time of Different Operations

With this finalized code, we run all three algorithms 95 times on the GPU and plot the execution time of each function. We run all three for 95 random polynomials in order of polynomial degree. That is, each iteration i of the 95-wide for-loop generates a random degree-i polynomial that is evaluated by each algorithm and timed.

We run the Horner's algorithm on a single thread in the GPU to act as a baseline. The other algorithms, Dorn's and Estrin's, are run as described in sections II-C and II-D respectively on multiple threads. Dorn's algorithm uses $k = \lfloor n/2 \rfloor$. We employ CUDA's Dynamic Parallelism and spawn GPU threads from a main GPU thread. When a function is called by the host, the main thread is spawned by the host and executes the algorithm. For example, in both Dorn's and Estrin's algorithms, a thread performs calculations first and then parallel threads are spawned. The parallel threads finish executing and the main thread returns the result. We choose to use a main GPU thread instead of the host to avoid CPU-GPU command execution time which can be substantial if frequent.

We evaluate the execution time and performance of all three algorithms where:

$$Performance_{alg} = -\frac{(execTime_{alg} - execTime_{baseline})}{execTime_{baseline}} \tag{5}$$

We vary the added latency by factors $\{1, \frac{1}{1000}, \frac{1}{100{,}000}, 0\}$ of the added SEAL latency to evaluate how the latency of individual operations impacts the execution time and performance of the three algorithms.

IV. Results and Discussion

Please refer to section A to see the figures discussed in the following sections.

A. Execution Time Evaluation

The plots for this section are shown in figure 2. With full added latency, we see that Estrin's algorithm achieves the lowest execution time, Dorn's algorithm achieves a worse but better than the baseline execution time, and Horner's algorithm achieves the worst execution time. With low degrees, they all start with around the same execution time but become very different as the degree increases. To better understand this observation, it is more useful to look at the performance plots in the next section. Horner's algorithm performs linearly as expected. Dorn's algorithm also performs linearly as expected. Estrin's algorithm performs nearly constant-like for the first 95 polynomial degrees, an incredible performance. Although it is constant-like, we see a non-linear curve forming suggesting that the performance is $O(\log(n))$ as expected (increasing the polynomial size would give a better idea of the real curve). This is very promising since polynomial degrees are usually less than 50 for PPML activation functions. With a polyomial degree of 15 having an x-bound of -100 and 100, a good PPML inference algorithm can be executed. Using Estrin's algorithm, we can have a low depth, 2ms algorithm, a very promising conclusion. For an even lower depth but 1ms more execution time, Dorn's algorithm can perform well, introducing a tradeoff hyperparameter between depth and performance.

With 1/1000 of the SEAL latency added, we see that all algorithms perform the same, with execution times three orders of magnitude lower than the first plot. This suggests that the performance of each algorithm is linear with respect to latency up to 1/1000.

With 1/100,000 of the SEAL latency added, we start to see worse performance for the two parallel algorithms. For example, for an-order 15 polynomial, it is better to use Horner's algorithm if lower execution time is important. The depth is an issue for Horner's algorithm which is why although Horner's algorithm executes better for polynomials of order-30 and less, it is suggested to use the other methods purely for the issue of depth. The execution times are not much worse either when comparing microseconds to microseconds.

Removing the added latency shows a different picture. We see that Horner's algorithm executes for much less time than the parallel algorithms for all of the polynomial degrees. The logarithmic nature of Estrin's algorithm is observed better in this example. It is important to note that as we keep increasing the degree, the execution times get closer for every algorithm suggesting that for very high order polynomials, the parallel algorithms still perform better than Horner's algorithm with Estrin's algorithm at the top.

It is important to show each algorithm's performance as a function of added latency since FHE libraries are constantly improving execution times. By improving the execution time of individual operations, whether parallelism is used or not to do so, these graphs show what PPML researchers can expect if using one of the three algorithms evaluated. In fact, since many algorithms are built off of the evaluated algorithms, these graphs also provide an idea of how each algorithm can affect the performance of a new algorithm. They provide a research direction.

B. Performance Evaluation

The plots for this section are shown in figure 3. Figure 3 shows how much better the parallel algorithms are than the baseline with performance measured as in (5). With full added latency, we see that Estrin's algorithm dramatically outperforms both of the other algorithms. Its performance is nearly linear with respect to Dorn's algorithm. This makes mathematical sense since both the baseline and Dorn's algorithm are linear in evaluation time, which is why they are a constant when compared to each other. In contrast, Estrin's algorithm is logarithmic and when compared to the baseline, it becomes linear. It is very intriguing that as the degree of the polynomial increases, the performance of Estrin's algorithm increases as well. For example, an order-15 polynomial can be evaluated in 250% better time using Estrin's algorithm than with using Horner's algorithm and in nearly 150% better time than Dorn's algorithm. It is also promising that Dorn's algorithm, with the lowest depth, always performs 100% better than Horner's algorithm. This suggests that one cannot go wrong if they choose to use Dorn's algorithm.

With 1/1000 of the SEAL latency added, we see that performance comes down drastically for Estrin's algorithm. Dorn's algorithm, however, is only 50% less performant than with full latency. This suggests that the true benefit of Estrin's algorithm is gained when latency is high. For the sake of comparison, with full added latency, an order-100 polynomial performs at 1300% whereas at 1/1000 added latency, the same polynomial performs at around 90%. That is nearly a 1/1000 decrease suggesting that varying latency causes linear performance. With these results, it is better for one to choose Dorn's algorithm over Estrin's algorithm for better depth, allowing them to perform more operations on CTs.

The remaining plots show that as we decrease latency, performance of both parallel methods decreases and Horner's algorithm triumphs. This suggests that the time to spawn threads becomes a factor in the execution time as addition and multiplication operation execution times decrease. However, one should still choose Dorn's algorithm when latencies become negligible solely for the depth benefits.

Considering both evaluation time and performance, with today's level of FHE compute ability, one should choose Estrin's algorithm since the performance margins are orders of magnitude apart while keeping the execution time reasonable (2ms for order-15). However, as latencies decrease and execution times become negligible, it is clear that the optimal algorithm is Dorn's algorithm because of the added depth benefits.

V. Conclusion

Fully Homomorphic Encryption is an exciting new field of security with many potential applications. This paper investigated FHE and machine learning, the intersection of which is an application called Privacy Preserving Machine Learning. We learned that activation functions become problematic in PPML due to no FHE support for non-linear operations. We saw that many researchers use polynomial approximations of activation functions instead of the activation functions themselves. We discovered a gap in the literature for evaluating polynomial functions in FHE. After researching many polynomial evaluation algorithms, we decided to evaluate three different algorithms applied to FHE polynomials: (1) Horner's algorithm, (2) Dorn's algorithm, and (3) Estrin's algorithm. We theoretically analyzed all three algorithms for depth and computational complexity. These algorithms strive to decrease the depth of an FHE circuit and utilize parallelism to better performance with respect to execution time. As a result, we evaluated the execution time and performance with respect to the baseline Horner's algorithm of all three algorithms. This paper shows that Estrin's algorithm achieves the best performance as a function of polynomial degree and lowest execution time out of the three algorithms with current FHE latencies based on the strongest FHE settings. It also shows that Dorn's algorithm becomes the optimal algorithm as latencies decrease, suggesting that it should be the algorithm of the future. For future work, we hope to run SEAL on a GPU to analyze the true performance of the algorithms rather than an execution-time-based identical version. It is also important to continue researching ways of parallelizing individual FHE operations to achieve lower latencies. One way is to develop hardware specifically designed for accelerating FHE operations. The list of algorithms we evaluated is non-exhaustive and future work could be to research and implement other polynomial evaluation algorithms.

Acknowledgment

The author would like to thank Professor Mark Jeffrey for giving him the opportunity to apply the knowledge gained in ECE1755 to his own research area. This work helped the author gain a head start on his Master's thesis research. The author would also like to thank Professor Jeffrey for always being so excited and inspiring when giving lectures. His energy and care for his students is unparalleled in the ECE department at the University of Toronto.

References

[1] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford, CA, USA, 2009, aAI3382729.

[2] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology – EUROCRYPT 2011*, K. G. Paterson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 129–148.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, Jul. 2014. [Online]. Available: https://doi-org.myaccess.library.utoronto.ca/10.1145/2633600

[4] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[6] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 201–210. [Online]. Available: http://proceedings.mlr.press/v48/gilad-bachrach16.html

[7] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," *CoRR*, vol. abs/1811.09953, 2018. [Online]. Available: http://arxiv.org/abs/1811.09953

[8] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, S. J. Jie, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "The alexnet moment for homomorphic encryption: Hcnn, the first homomorphic CNN on encrypted data with gpus," *CoRR*, vol. abs/1811.00778, 2018. [Online]. Available: http://arxiv.org/abs/1811.00778

[9] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Advances in Cryptology – CRYPTO 2018*, H. Shacham and A. Boldyreva, Eds. Cham: Springer International Publishing, 2018, pp. 483–512.

[10] Y. Fan, J. Bai, X. Lei, Y. Zhang, B. Zhang, K.-C. Li, and G. Tan, "Privacy preserving based logistic regression on big data," *Journal of Network and Computer Applications*, vol. 171, p. 102769, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804520302435

[11] Y. Aono, T. Hayashi, L. Phong, and L. Wang, "Privacy-preserving logistic regression with distributed data sources via homomorphic encryption," *IEICE Trans. Inf. Syst.*, vol. 99-D, pp. 2079–2089, 2016.

[12] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 315–323. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a.html

[13] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *CoRR*, vol. abs/1711.05189, 2017. [Online]. Available: http://arxiv.org/abs/1711.05189

[14] M. Abramowitz, *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables,*. USA: Dover Publications, Inc., 1974.

[15] "Microsoft SEAL (release 3.6)," https://github.com/Microsoft/SEAL, Nov. 2020, microsoft Research, Redmond, WA.

[16] W. G. Horner, "A new method of solving numerical equations of all orders, by continuous approximation," *Philosophical Transactions of the Royal Society of London*, vol. 109, pp. 308–335, 1819. [Online]. Available: http://www.jstor.org/stable/107508

[17] W. S. Dorn, "Generalizations of horner's rule for polynomial evaluation," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 239–245, 1962.

[18] A. Kiper, "Modified dorn's algorithm with improved speed-up," in *Applied Parallel Computing Industrial Computation and Optimization*, J. Waśniewski, J. Dongarra, K. Madsen, and D. Olesen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 433–442.

[19] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[20] I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, 1971, pp. 132–139.

[21] G. Estrin, "Organization of computer systems: The fixed plus variable structure computer," in *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, ser. IRE-AIEE-ACM '60 (Western). New York, NY, USA: Association for Computing Machinery, 1960, p. 33–40. [Online]. Available: https://doi.org/10.1145/1460361.1460365

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[24] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 11.3.58," 2021. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[25] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans*, E. Pasalic and L. R. Knudsen, Eds. Cham: Springer International Publishing, 2016, pp. 169–186.

[26] V. Group, "vernamlab/cuFHE," Apr. 2021, original-date: 2018-03-14T18:07:47Z. [Online]. Available: https://github.com/vernamlab/cuFHE

[27] B. Opanchuk, D. Pierre, and M. Fan, "nucypher/nufhe," Apr. 2021, original-date: 2018-04-08T05:17:17Z. [Online]. Available: https://github.com/nucypher/nufhe

[28] "Intel HEXL (release 1.0)," https://arxiv.org/abs/2103.16400, Mar. 2021.

Appendix

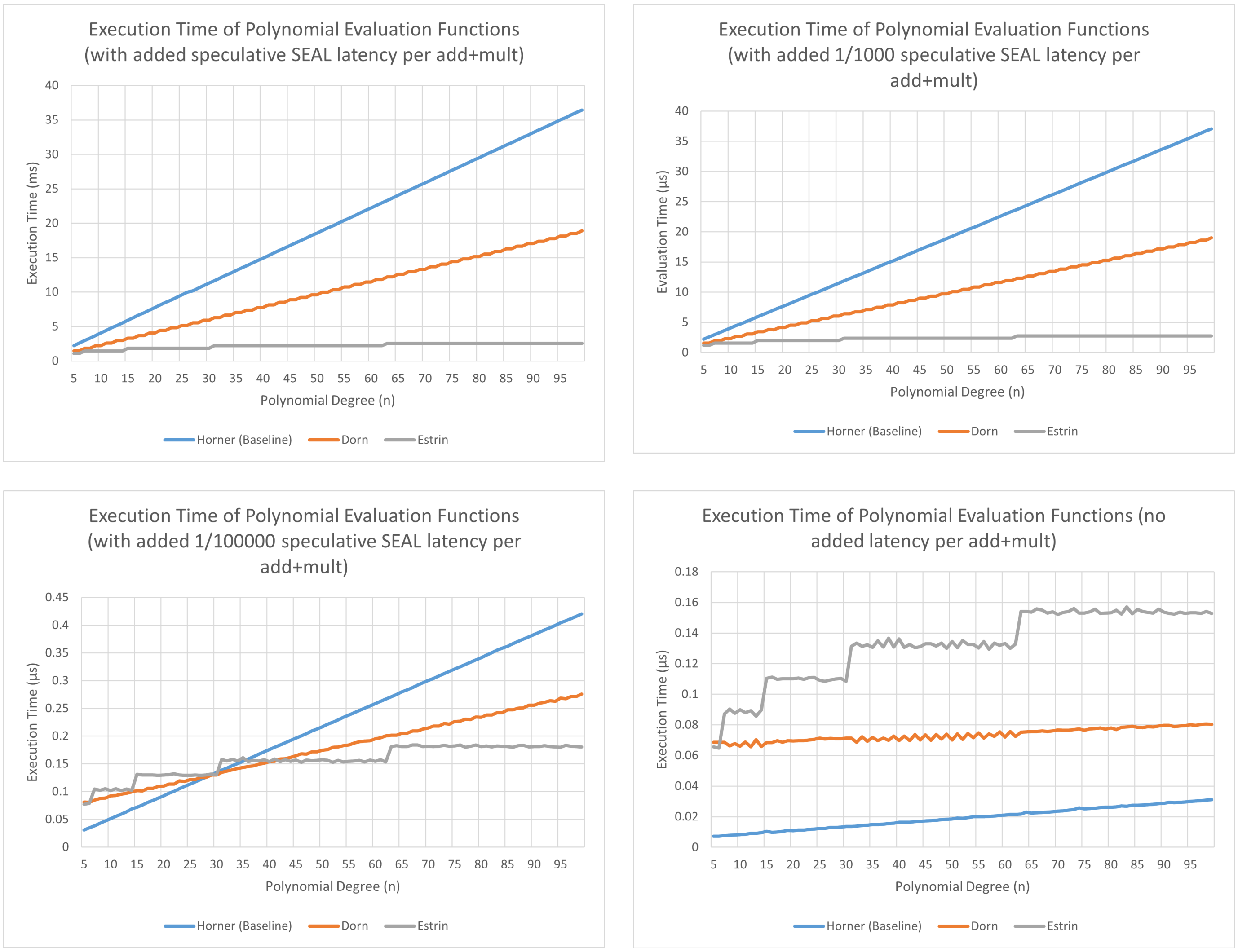


Fig. 2: Execution time of algorithms as amount of added latency decreases by labelled factors

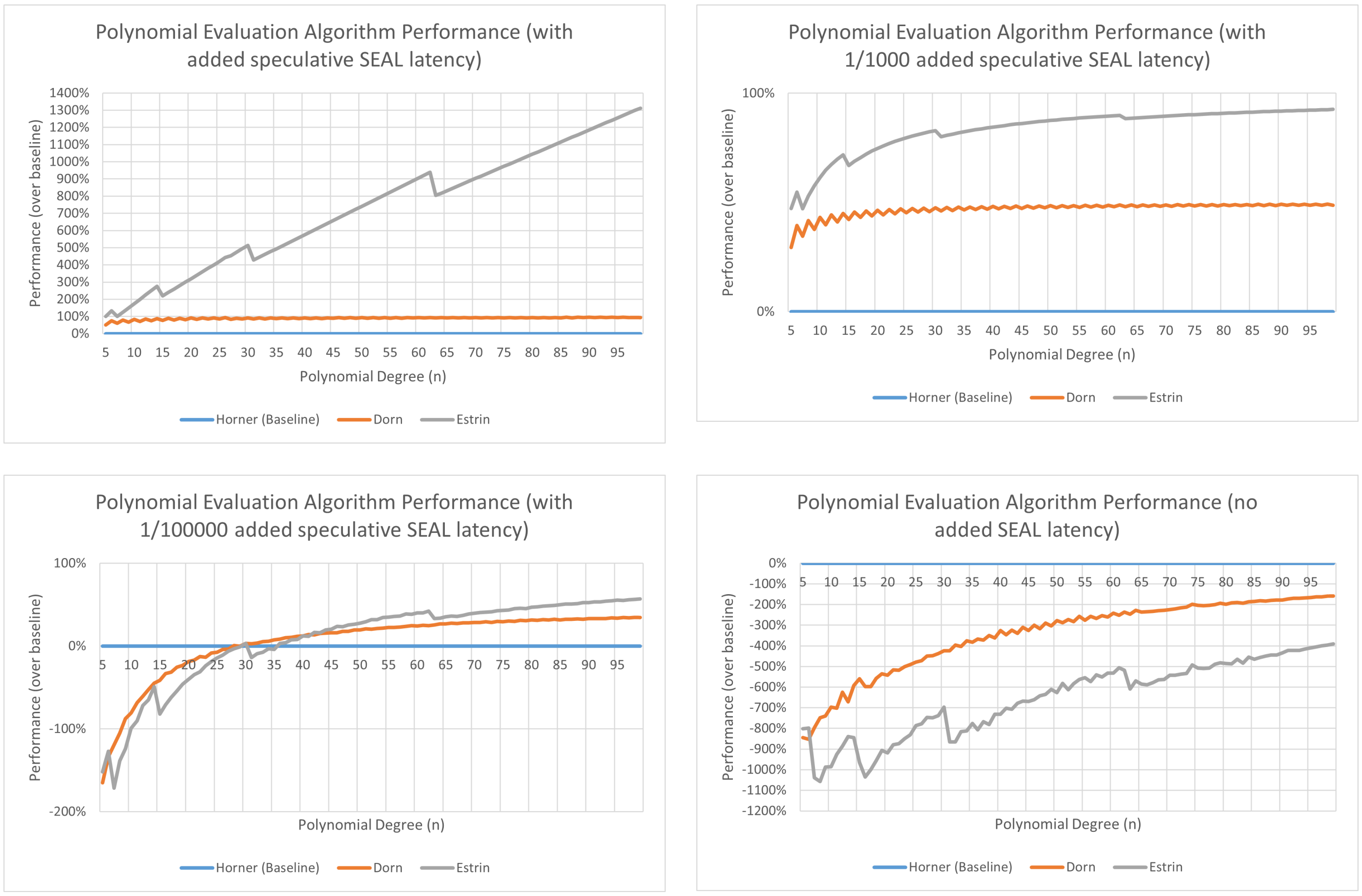


Fig. 3: Performance of algorithms with respect to baseline as amount of added latency decreases by labelled factors