Enhancing the Privacy of Machine Learning via faster arithmetic over Torus FHE

Marc Titus Trifan
Department of Computer Science
University of California, Irvine
Irvine, USA
mtrifan@uci.edu

Alexandru Nicolau
Department of Computer Science
University of California, Irvine
Irvine, USA
nicolau@ics.uci.edu

Alexander Veidenbaum
Department of Computer Science
University of California, Irvine
Irvine, USA
alexv@ics.uci.edu

***Abstract*—The increased popularity of Machine Learning as a Service (MLaaS) makes the privacy of user data and network weights a critical concern. Using Torus FHE (TFHE) [1] offers a solution for privacy-preserving computation in a cloud environment by allowing computation directly over encrypted data. However, software TFHE implementations of cyphertext-cyphertext multiplication needed when both input data and weights are encrypted are either lacking or are too slow. This paper proposes a new way to improve the performance of such multiplication by applying carry save addition. Its theoretical speedup is proportional to the bit width of the plaintext integer operands. This also speeds up multi-operand summation.**

A speedup of 15x is obtained for 16-bit multiplication on a 64-core processor, when compared to previous results. Multiplication also becomes more than twice as fast on a GPU if our approach is utilized. This leads to much faster dot product and convolution computations, which combine multiplications and a multi-operand sum. A 45x speedup is achieved for a 16-bit, 32-element dot product and a ∼30x speedup for a convolution with a 32x32 filter size.

***Index Terms*—Fully Homomorphic Encryption, Torus FHE, Gate Bootstrapping, CPU Parallelism, Carry Save Addition**

I. Introduction

Homomorphic Encryption (HE) allows operations to be performed directly on encrypted data. A number of HE schemes have been developed. They vary in the type of operations supported (integer, boolean, real), the complexity of theoretical problems they are based on (LWE, ring LWE, etc), and whether they are leveled or fully HE. The most used schemes include BFV [2] and BGV [3] for integer arithmetic, CKKS [4] for real arithmetic, and TFHE [1], GSW [5] and FHEW [6] for boolean arithmetic. There are multiple library implementations of these schemes, like Microsoft SEAL [7], IBM HELib [8] OpenFHE [9] or TFHE-specific only implementations, like TFHE [10] or Concrete [11].

Leveled HE schemes allow only a fixed number of operations to be performed, after which point a ciphertext may become undecryptable. An operation called bootstrapping was introduced in [12] to allow a computation to continue without limitations. However, bootstrapping is a very time consuming operation compared to FHE arithmetic operations (which are already very expensive).

FHEW and TFHE introduced an inexpensive bootstrapping operation, which TFHE applies after every boolean operation it supports. This makes it a very attractive scheme compared to others that require occasional but very expensive bootstrapping.

At the level of integer operations, the fact that TFHE only implements boolean operations is quite detrimental to performance. This is because a B-bit integer addition and multiplication are very sequential operations when implemented using boolean gates in software. An addition, for instance, computes and propagates carry through all the bit positions sequentially, thus requiring $O(B)$ gate delays. Fast carry-lookahead schemes used in hardware implementations are not effective in software due to the growth in the number of gates to evaluate. Similarly, fast hardware array multipliers cannot be effectively implemented in software as this would require massive parallelism. For instance, just forming all of the partial products to be summed at once would require B^2 TFHE AND operations. A software implementation of the multiply thus typically uses a shift-and-add algorithm resulting in $O(B^2)$ gate delays.

This paper shows how to exploit parallelism to improve the performance of integer operations over TFHE. The parallelism is created by performing a carry-save addition (CSA) in the case of $\sum_0^{K-1}$ operators. The CSA version used only propagates the carry by one bit position, which makes a B-bit CSA fully parallel in the number of bits. Integer multiplication, when computed as $P = \sum_{i=0}^{B-1}(A \times c_i \times 2^i)$ for $A \times C$, where c_i is a bit of C, can thus exploit this type of parallelism.

The proposed approach can be utilized for neural network inference computations in the cloud. An ML system may require that both the input data and the weights are encrypted to preserve the privacy of both a user and a neural network provider. This work thus targets integer arithmetic operations where both operands are encrypted, a case that is either not currently supported or is inefficient.

Overall, this paper makes the following contributions:

1) It uses parallel carry-save addition to speed up integer multiplication
2) It speeds up a summation of multiple operands using carry-save addition and applies this in the context of dot product and convolutional neural network computations
3) It explores ways to parallelize operations and discusses ways to map them to the multiple cores available

4) It demonstrates significant speedups (45x and 30x for the dot product and the CNN computations, respectively) for 16-bit operands on a 64-core processor.

II. TFHE

The TFHE library [1] uses a binary representation on a torus of real numbers considered modulo 1, $T = \mathbb{R}/\mathbb{Z}$, where the plaintext space is represented by [-1/2, 1/2], with the binary values 0 and 1 represented as $-1/8$ and $1/8$. The supported homomorphic operations are logical gates (NOT, AND, OR, XOR, etc). Each gate evaluation involves a bootstrapping operation.

A ciphertext in the TFHE domain is represented by a pair (a, b) with a a vector of length n, concatenated together with a torus element b, such that $b = a \cdot s + e$, where s is a vector of length n and e is a small error term called noise, being sampled from a Gaussian distribution.

The main advantages of the TFHE library are fast bootstrapping, lower polynomial and coefficient sizes, but at the expense of constant bootstrapping.

An example of gate evaluation is shown below.

$$NOT(a) = \sim a$$
$$AND(a, b) = a + b - 1/8 \gtrless 0$$
$$OR(a, b) = a + b + 1/8 \gtrless 0$$
$$XOR(a, b) = 2(a + b) + 1/4 \gtrless 0$$

The purpose of the bootstrapping is to evaluate the corresponding condition, and to further reset the resulting message either as a fresh encryption of 0 or 1, with low noise level. Note that, for bootstrapping to work, the operands a and b must be fresh ciphertexts.

The NOT gate comes as a free gate in TFHE, since no bootstrapping is involved. Another useful gate in the TFHE world is the possibility of evaluating multiplexers, i.e. a MUX gate, which allows the switching of one of the two inputs based on a selector. The MUX gate introduces additional versatility to the FHE world which would be hard to implement using other more traditional schemes.

For 128-bit security, TFHE uses a LWE polynomial size of 630, a RLWE polynomial size of N=1024, where each integer coefficient is considered modulo q=2^{32}.

TFHE uses the notion of lookup tables to bootstrap a ciphertext. The bootstrapping operation will obtain the desired element from an unknown (encrypted) position p in the lookup table as a result of bootstrapping. The gate bootstrapping in TFHE is used to evaluate binary gates, but can be generalized as a general function evaluation only by changing the lookup table to be evaluated.

A new(er) library called Concrete [11] was built on top of the TFHE library. Both libraries use the same basic concepts, but the Concrete library replaces gate bootstrapping with functional bootstrapping. For this paper we use the TFHE library available at [10].

III. Integer arithmetic on top of TFHE

Integer arithmetic is supported by default in schemes like BGV. However, these schemes employ expensive ciphertext maintenance operations, like modulus switching or bootstrapping, as measures of noise control.

After a fixed number of multiplications, bootstrapping has to be performed in order to ensure the accuracy of the results. This incurs a high overhead, in particular while trying to evaluate a large number of multiplications over the ciphertext domain.

Integer arithmetic can also be built on top of binary FHE schemes, such as GSW or TFHE. This will clearly increase the time required to perform integer ciphertext operations. However, the TFHE scheme offers a performance advantage due to inexpensive bootstrapping, smaller polynomial size, and 32-bit polynomial coefficients. All of these may lead to improved performance when compared with the BGV scheme where several expensive bootstrapping operations may be needed.

Our approach can be used without any modification for fixed-point arithmetic. Fixed-point values can be used as weights for different AI or ML purposes, and in our case, their usage will not incur any additional performance overhead compared to using integer values.

IV. Integer multiplication

The goal of this paper is to increase the CPU-level parallelism of operations built on top of the TFHE scheme. Our focus is on ciphertext-ciphertext operations, and in particular, on multiplication.

We assume an operation on two n-bit integers, a and b, with their binary representation expressed as $a = [a_0, a_1, .., a_{n-1}]$, $b = [b_0, b_1, .., b_{n-1}]$.

The TFHE library provides us with boolean operations, which can be used to implement an (single-bit) add functionality. This, in turn, allows an n-bit add function to be built. Such software adder can then be utilized to perform integer multiplication. Next, we consider approaches to do this.

A standard approach to implement multiplication of integers using binary representation is to use a naive schoolbook algorithm. This is a shift-add algorithm. It forms a partial product by multiplying a bit b_i of the multiplier with every bit a_j of the multiplicand. The partial product is then added to the final result.

The major drawback of this algorithm is that the addition operation has to be performed sequentially because of the carry propagation.

[13] proposed an approach to speed up multiplication that used to the sequential addition in the TFHE context by introducing limited parallelization. It formed partial products in parallel and then added them using a sum reduction operation (in OpenMP [14]). The addition operation performed within reduction still used the sequential carry propagation. As shown in Sec. V, this approach has a rather limited speedup.

The idea behind carry-save addition is to compute a Sum and a Carry-out for each bit position without propagating the

Carry-out to the adjacent bit position. Therefore, it is not producing the correct sum for each bit position, but it produces the Sum and Carry-out bits in each position independently (i.e. it is fully parallel). In fact, the carry is propagated one bit position at a time as Sum$_i$ is computed using the Carry-out$_{i-1}$.

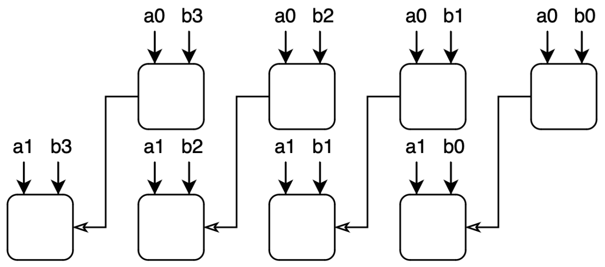
Carry-save addition cannot therefore speed up the addition operation, but it can speed up repeated additions or a summation, possibly requiring a final add with carry propagation. This is exactly the case in the multiply operation.

Each addition step produces the final value for the least significant bit, which is shifted into the result array. An n-bit result is computed after n such addition steps and does not require further carry propagation. A 2n-bit result would require the carry to be propagated over the n high-order bit positions.

The above approach seems to have been first described in [15], per reference in [16]. It exposes just the right amount of parallelism for our target system. The Wallace or Dadda multipliers [16], [17] expose even more parallelism, generating all partial products at once and using a tree of carry-save adders. However, this requires many more gates (and thus cores) to be evaluated. This work assumes a rather limited number of cores per processor (64 or fewer) and, for common plaintext operand sizes of 8- to 32-bits, is unable to utilize the Wallace or Dadda multiplier tree parallelism.

Even though hardware CSA has been proposed, its application in the TFHE context is new and a parallel implementation to achieve speedup is non-trivial.

Fig. 1. Two iterations of carry save adder



The $i-th$ step of our multiplication computes and stores in parallel the result of multiplication of a_i and b_j, $j = 0...n-1$. Next, "result" is computed in parallel (over j), followed by the parallel computation of "carry".

In other words, when computing the result as $a_i \oplus b_j \oplus c$, the carry is not generated in the current iteration, but is used as generated by the "carry" step of the previous iteration. In this way, both the "result" and the "carry" steps are fully parallel, having dependencies just on the last stage of multiplication with bit a_{i-1}.

Suppose that we want to perform the addition of bit r, representing one of the accumulator bits, with bit p, representing the product of two bits.

The "result" step will compute (in parallel for all bits):

$$r_{new} = r \oplus p \oplus c_{prev} \tag{1}$$

The "carry" step will compute (in parallel for all bits) the new carry to be used at the new iteration, as:

$$c_{out} = r \wedge p + p \wedge c_{prev} + r \wedge c_{prev} \tag{2}$$

Figure 1 represents visually two consecutive steps of the carry save multiplication. The carry is computed in the current step, but its addition to the final result is postponed up until the next step.

There are several optimizations one can employ for this implementation. For example, one can reuse a gate from the previous result computation when calculating the next carry.

This approach uses 5 (software) TFHE gates for computing the carry, and we can also save an additional gate (and the respective involved bootstrapping) by using the fact that one of the gates was previously computed at the result step. It was observed that memory constraints allow storing the result from the first computation such that it can be reused in the partial carry computation.

For this purpose, one can rewrite the carry expression as the following equivalent expression:

$$c_{out} = ((r \oplus p) \wedge c_{prev}) \vee (r \wedge p) \tag{3}$$

We observe that the operation $a_i \oplus b_i$ is already computed during the next result computation, which is done in parallel right before the parallel computation of the carry, so the partial xor can be stored into the memory and reused.

Bootstrapping still consumes a lot of time in each TFHE gate evaluation. One can reduce the number of bootstrappings per 1-bit CSA by using ternary gates.

A. CSA + Ternary gates

[18] introduced ternary gates to the TFHE scheme, in particular an XOR3 and a 2OF3 gate. [19] revisits the idea and applies it to integer addition over TFHE. The purpose of these additional gates was to perform the computation of a full adder while reducing the number of bootstrappings.

An XOR3(a,b,c) gate is defined as the XOR of the three inputs, and a 2OF3 gate outputs 1 if and only if at least two out of the three inputs are set to 1.

For example, computing a new sum requires 2 bootstrappings using 2 XOR gates, but requires just 1 bootstrapping using the XOR3 gate. Similarly, computing the new carry requires 3 bootstappings using binary gates, but only 1 when using the 2OF3 gate.

The implementation of the new ternary gates is as follows:

$$XOR3(a, b, c) = -2(a + b + c) \gtrless 0 \tag{4}$$

$$2OF3(a, b, c) = (a + b + c) \gtrless 0 \tag{5}$$

In this way, the number of bootstrappings needed when evaluating a full adder is reduced from 5 to 2, considerably improving the performance of the scheme. However, as noted in [19], the ternary gates require larger parameters to allow the evaluation of the combined gate without exceeding the level of noise permitted for decryption. The default parameter set of

TFHE, which allows 128 bit security, satisfies the requirements and can be used in our scenario.

In summary, we combine the use of ternary gates and carry save multiplication.

V. Performance evaluation

This section presents the performance results of our scheme and compares them with the results in [13]. The code available in [20] was used. It also shows the additional speedup from using ternary gates [19]. The performance was evaluated on a system with two sockets, with the AMD EPYC 7742 processor with 64 cores running at 2.25 Ghz and NUMA memory [21]. All results on the AMD system are on a single socket.

The original TFHE library with 128 bit security [1] was used. It was modified to incorporate ternary gates [19], [22]. A carry-save adder was implemented using either the original TFHE two-input gates or ternary gates. In the latter case, the sum and carry incur a one-gate delay.

A multiply for two n-bit numbers produces a 2n-bit result. However, it is not always needed in a program performing n-bit computation. One can ignore the upper n bits of the 2n-bit result and just use the lower n bits. The n-bit result is obtained in this case with only carry-save adds.

A. *Integer multiplication*

Let us begin with integer multiplication performed as a sequence of carry-save adds and shifts. Each step produces one bit of the product. It was evaluated by a varying the number of threads used and compared with [13]. Note that the algorithm in [13] cannot profitably utilize more than 4 to 8 threads for the operand sizes used as it only performs a reduction in parallel.

Table I shows the multiply execution times on the AMD processor. These are n-bit results only, i.e. they do not require a full add at the end. The best execution time in each row is shown in boldface. The table also shows the best speedup obtained for each case over the [13] results (re-executed on the AMD processor). A 10.63x speedup is achieved over results in [13] for 8-bit multiplication with both CSA and ternary gates. The speedup is 14.76 for 16-bit operands. It then drops to 8.77 for 32-bit operands. This is in part due to our multiplication utilizing all 64 cores, which increases communication costs between tiles.

These results can also be compared to the fully sequential execution, e.g. executing our CSA code on one thread. The parallel CSA+ternary approach obtains a speedup of 10.3x, 19.8x and 15.8x, respectively, for 8-, 16- and 32-bit multiply. Parallel CSA speedups are approximately one-half of the CSA+ternary.

Thread distribution across the system cores plays an important role with respect to the performance of the multiplication. The results shown set the affinity of the OpenMP context to $scatter$, meaning that the threads are distributed evenly across the system.

Sequential timing for the addition as implemented in [13] but on the AMD architecture, is as following: 0.67s, 1.36s and 2.89s for 8, 16 and 32-bits, respectively.

The many-core CPU results for multiplication are also compared to the best previous known GPU results as presented in [13], which are for a non-Karatsuba implementation of the multiplier. We ported the original GPU code to our more modern GPU system and compared it to our "CSA+ternary" CPU approach. Ours is 3.41 and 2.03 times faster for 16-bit and 32-bit multiplication, respectively. Of course, our "CSA+ternary" can also be applied on a GPU to improve performance. We demonstrate this by modifying the GPU implementation from [13].

Table II shows the execution times of the "CSA+ternary" multiplier on GPU. The GPU system in both cases was NVIDIA TITAN X (Pascal), with 12GB of memory. The TFHE library ported on GPU by [20] was used unmodified for the "GPU [13]" version. The modified library added support for ternary gates and the "CSA+ternary" algorithm was implemented on top of it. This resulted in a speedup of 2.15, 2.74, and 2.92x for 8-, 16-, and 32-bit multiplication, respectively. These results are still slower than our CPU results for 8- and 16-bit multiply, but faster for the 32b multiply by 30%. Several optimizations can still be applied for the GPU, at the level of TFHE gate implementation, but this is outside of the scope of the paper.

Furthermore, many important computations which combine multiplications and a multi-operand sum can benefit from the carry-save approach. The efficacy of the proposed optimizations is evaluated next for two algorithms, the dot product and the Convolutional Neural Network (CNN) inference convolution kernel.

B. *Dot Product*

The dot product of two n-element vectors a and b, is defined as $c = c + a_i * b_i$, with $i = 0...n-1$, where a_i and b_i are B-bit integers representing the i-th element of the vectors.

The sequential time can be estimated as $T_{seq} = n * T_{mul} + (n-1) * T_{add}$, where T_{mul} and T_{add} are the multiplication and addition times for B-bit integers.

Our approach uses CSA with the ternary gates and consists of B CSA additions using B threads. In addition to this, the $n-1$ additions within the dot product can now be performed as CSAs and combined with the multiplication CSAs. Thus, all the additions required in the dot product are now performed as CSAs, assuming that only the B-bit result is required.

The dot product performance for 16- and 32-bit operands and vector sizes of 16, 32 and 64 are shown in Table III. The sequential times are based on T_{mul} of the sequential CSA multiplication time. Because of the naive multiplication version computing a $2n$-bit result, the sequential carry save version (without ternary gates) is a more fair comparison in terms of speedups. However, when compared to the naive version, a $\sim$70x speedup is obtained for the 16-bit, 32 element dot product

The CSA' version performs the addition step together with the CSA multiplier but uses only one group. The $\|$ CSA version uses 4 thread groups at the outer level, in which each group uses 16 threads, for a total of 64 threads.

TABLE I
TIMING (SEC) FOR INTEGER MULTIPLICATION ON AMD EPYC, W/O FINAL FA

Threads	Seq	2	4	8	16	32	64	Speedup
8 bits								
[13]	12.75	7.33	**6.70**	12.42	-	-	-	1x
CSA	6.54	2.92	2.11	**1.27**	1.30	1.28	1.27	5.27x
CSA+ternary	3.45	1.52	1.00	**0.63**	0.64	0.64	0.63	10.63x
16 bits								
[13]	46.99	30.35	**20.82**	27.94	-	-		1x
CSA	27.93	13.6	8.27	4.64	**2.70**	2.91	2.85	7.71x
CSA+ternary	14.00	7.82	3.97	2.42	**1.41**	1.45	1.42	14.76x
32 bits								
[13]	189.39	121.60	71.21	**62.66**	192.73	-	-	1x
CSA	113.46	60.16	32.03	22.48	22.15	17.73	**16.68**	3.75x
CSA+ternary	56.95	29.76	16.17	11.86	9.82	8.45	**7.14**	8.77x

TABLE II
TIMING (SEC) FOR INTEGER MULTIPLICATION ON GPU

Bit length	GPU [2]	CSA+ternary	Speedup
8 bit	1.64	0.76	2.15x
16 bit	4.81	1.75	2.74x
32 bit	14.53	4.97	2.92x

The speedups of the CSA' and || CSA approaches with respect to the purely sequential one are presented in Table IV. The CSA' version obtains a speedup of ∼17x for the 16-element dot product, for both 16 and 32-bit operands, and ∼ 13x for 64 elements.

Parallelizing the 64-element dot product on top of the CSA' version obtains a speedup of 3.13x for 16-bit elements, and one of 3.33x for 32-bit elements. We attribute the difference to the final sequential add which has to be performed in the parallel version, as well as to the hardware structure of the system, which makes the multiplication time slightly higher than the parallel one. We observe increasing speedups with the increase in vector length for the || CSA approach compared to the CSA' version.

Compared to a purely sequential approach, we obtain a speedup of 45.36x for 32-element, 16-bit dot product and 48.82x for 16-element, 32-bit dot product.

C. CNN Evaluation

Convolutional neural networks (CNNs) are used in Deep Learning, especially for image processing. The kernels determine a weighted sum of the input feature values. The convolution layers consume most of the inference time.

The kernel is used as a sliding window on top of the actual feature map. For each particular case, the corresponding elements from the feature map and from the kernel are multiplied together and summed up to form one output result. We consider the case where the input has Q feature maps, and there will be R output feature maps obtained. Each of the feature maps, and the convolution kernel are two-dimensional inputs.

A typical convolution code is shown in Algorithm 1. The first two loops iterate over each input/output feature map. The next two loops iterate over the feature map, while the last two go over the convolution kernel dimensions.

Algorithm 1 Convolution - Version 1

```
Input: $X$ - 3D input, $W$ - convolution kernel
Output: $Z$- 3D result of the convolution
1: for $r = 0$ to $R$ {output feature map} do
2:   for $q = 0$ to $Q$ {input feature map} do
3:     for $m = 0$ to $M$ {row in feature map} do
4:       for $n = 0$ to $N$ {col in feature map} do
5:         for $k = 0$ to $K${row in conv. kernel} do
6:           for $l = 0$ to $L$ {col in conv. kernel} do
7:             Z[r][m][n]+=W[r][q][k][l]*X[q][m+k][n+l]
8:           end for
9:         end for
10:       end for
11:     end for
12:   end for
13: end for
14: return $Z$
```

This implementation is referred to as "Version 1". Note that the actual computation involves a product between kernel values and feature map values, which are added into the final result. All the multiplications are accumulated into the same result, which makes the computation similar to our dot product approach.

We also propose a different version of the algorithm, referred to as a "Version 2". It is obtained by interchanging the 2 innermost loops with the 2 middle loops. In this way, while the correctness is satisfied, we accumulate into different elements of the output. This implementation greatly increases the parallelism of the algorithm.

Only the two innermost for loops were timed for each version because the rest of the computation is sequential and follows the same memory access and computation pattern.

A sequential version of the two innermost loops executes in $T_{seq} = M * N * T_{mul} + M * N * T_{add}$, where T_{mul} and T_{add} are the average multiplication and addition times for n-bit integers, and M and N are the number of rows and columns in the feature map.

TABLE III
TIMING (SEC) FOR DOT PRODUCT

Vector	16 bit			32 bit		
Length	Seq	CSA'	‖ CSA	Seq	CSA'	‖ CSA
16	467.28	25.36	**11.22**	1858.71	109.44	**38.07**
32	935.92	61.16	**20.63**	3720.31	316.68	**103.28**
64	1873.20	148.01	**47.14**	7443.51	570.28	**170.99**

TABLE IV
SPEEDUP FOR DOT PRODUCT WRT SEQUENTIAL

Vector	16 bit		32 bit	
Length	CSA'	‖ CSA	CSA'	‖ CSA
16	18.42x	41.64x	16.97x	48.82x
32	15.30x	45.36x	11.74x	36.02x
64	12.65x	39.73x	13.05x	43.53x

TABLE V
TIMING (SEC) FOR CNN EVALUATION - VERSION 1

Kernel	16 bit			
Size	Seq	CSA	‖ CSA	Speedup
5x5	732.25	93.79	24.34	30.08x
7x7	1435.21	198.74	40.61	35.34x

The same approach as the described above for the dot product is used here. We parallelize the innermost loop, splitting the computation in line 8 into g groups, each group computing N/g elements. The accumulation into the result is in this way fully parallel.

The Version 1 "‖ CSA" implementation employed g parallel groups at the outer loop level. The inner for loop is completely replaced by a CSA accumulation of L elements. As in the case of the dot product, another sequential addition step over the partial results is required at the end of the computation.

The Version 2 "‖ CSA" implementation can be fully parallel, where each basic operation is a CSA multiplication and accumulation of the final result, performed in one go. No other final adder step is required.

Utilizing $t = 16$ threads inside the multiply and $g = 4$ groups at the outer level yields the best results for the architecture used with 16-bit operands.

The results for the CNN kernel are shown in Table V, while Table VI presents the Version 2 implementation. The sequential time was computed in the same way as in the Dot Product case. The CSA version used CSA multipliers instead of the regular ones. The ‖ CSA approach is similar to the dot product one, when using 4 thread groups at the outer level, and 16 threads for each CSA multiplier.

Version 1 obtains a speedup of 30.08x and 25.34x for 5x5 and 7x7 kernels, respectively. Version 2 obtains a 35.41x and 30.52x speedup for 16x16 and 32x32 filter size, respectively.

TABLE VI
TIMING (SEC) FOR CNN EVALUATION - VERSION 2

Filter	16 bit			
Size	Seq	CSA	‖ CSA	Speedup
16x16	7498.24	1030.92	211.72	35.41x
32x32	29992.96	4113.68	982.65	30.52x

We got similar speedups for the two versions. However, Version 1 operates on kernel sizes, which are typically small, but in case of Version 2 the parallelism can be explored up to the size of the input. The goal of this paper was to optimize single node performance, but the ideas can be extended to multi-node systems (e.g. using MPI) for further significant speedup, in particular when utilizing Version 2.

VI. PRIOR WORK

Several approaches to build integer arithmetic on top of the TFHE scheme have been investigated.

Google Transpiler: The work in [23] describes a transpiler from high level C++ (integer operations) to C++ code written with the use of the TFHE library (binary gates). They use Google's XLS Boolean Circuit Optimizer to perform optimizations of the circuit before being translated to TFHE gates. However, this operation does not fully generalize to the particularities of FHE.

Other external libraries/compilers for optimizing TFHE boolean circuits: The work in [24], [25] describes a general optimizer for TFHE code. The input program has to be either written already with the use of the TFHE library primitives or by using a different high-level domain specific language.

Parallel CPU or GPU implementations: [13] describes the use of parallelism on CPU to improve integer addition and multiplication on top of the TFHE scheme. Out of tried approaches, the best speedup on CPU was obtained when using an OpenMP reduction, which uses global shared memory to store partial results at each multiplication iteration. The CPU-based multiplication obtains a $\sim$ 2.5x speedup with respect to the sequential approach. The paper also describes a GPU implementation. We also mention a few other GPU implementations of the TFHE scheme, together with the integer arithmetic implementation, such as [26] or [27].

Ternary gates: The work in [18], [19] speeds up TFHE integer arithmetic by introducing ternary or 3-input gates. The main advantage of this is the reduction in the number of required bootstrapping operations, which dominate the evaluation time of a gate. For instance, an XOR of three inputs normally requires two 2-input (regular) XOR gates, that each performs a bootstrapping. A 3-input XOR performs a bootstrapping once. The paper also introduces other optimizations, but with a focus to plaintext-ciphertext multiplication. Our work focuses on ciphertext-ciphertext multiplication.

A. Acknowledgements

The authors would like to thank Dr. Rosario Cammarota for fruitful personal discussions. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. The Expanse System at the San Diego Supercomputing Center was used, through allocation id TG-ASC150024.

VII. Conclusion

This paper showed how to speedup TFHE gate bootstrapping by utilizing parallelism. It also showed how to significantly increase the parallelism in integer multiplication using the Carry Save Addition. Significant performance improvement on many-core processors was achieved in both cases. Furthermore, applications with summation operations, such as dot products and convolutional neural networks, gained additional performance because summations can also utilize the Carry Save Addition. The exploitation of parallelism required significant program analysis, modification, and tuning to achieve good performance. Careful attention to thread affinity, nested parallelism, NUMA control (where applicable), etc was also required. Your mileage will vary depending on processor and compiler used. Future work will include better understanding of the nested parallelism issues and achieving further speedup by utilizing both parallel integer operations and parallel bootstrapping.

References

[1] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, Jan. 2020.

[2] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12, Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325.

[4] 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.

[5] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 75–92.

[6] L. Ducas and D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640.

[7] Microsoft, *Microsoft SEAL*, https://github.com/Microsoft/SEAL, Microsoft Research, Redmond, WA., Mar. 2022.

[8] S. Halevi and V. Shoup, *Design and implementation of HElib: A homomorphic encryption library*, Cryptology ePrint Archive, Paper 2020/1481, 2020. [Online]. Available: https://eprint.iacr.org/2020/1481.

[9] A. A. Badawi, J. Bates, F. Bergamaschi, *et al.*, *OpenFHE: Open-Source Fully Homomorphic Encryption Library*, Cryptology ePrint Archive, Paper 2022/915, 2022.

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, *TFHE: Fast Fully Homomorphic Encryption Library over the Torus*, https://github.com/tfhe/tfhe, 2020.

[11] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, *CONCRETE: Concrete Operates on Ciphertexts Rapidly by Extending TfhE*, 2020.

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

[13] T. Morshed, M. Aziz, and N. Mohammed, "CPU and GPU Accelerated Fully Homomorphic Encryption," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2020, pp. 142–153.

[14] OpenMP Architecture Review Board, *OpenMP application programming interface, version 5.0*, Nov. 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[15] J. Robertson, *Theory of Computer Arithmetic Employed in the Design of the New Computer at the University of Ilinois*, Ann Harbor, Jun. 1960.

[16] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.

[17] L. Dadda, *Some schemes for parallel multipliers*, Alta. Frequenza, Vol. 34, 1965.

[18] K. Matsuoka, Y. Hoshizuki, T. Sato, and S. Bian, "Towards better standard cell library: Optimizing compound logic gates for tfhe," in *Proceedings of the 9th on Workshop on Encrypted Computing, Applied Homomorphic Cryptography*, ser. WAHC '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 63–68.

[19] J. Klemsa and M. Önen, "Parallel Operations over TFHE-Encrypted Multi-Digit Integers," in *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '22, Baltimore, MD, USA: Association for Computing Machinery, 2022, pp. 288–299.

[20] T. Morshed, M. Aziz, and N. Mohammed, *Cpu and gpu accelerated fully homomorphic encryption*, https://github.com/toufique-morshed/CPU-GPU-TFHE, 2020.

[21] J. Towns, T. Cockerill, M. Dahan, *et al.*, "XSEDE: Accelerating Scientific Discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, Oct. 2014.

[22] J. Klemsa, *Parmesan*, https://github.com/fakub/parmesan/tree/param-sets, 2021.

[23] S. Gorantala, R. Springer, S. Purser-Haskell, *et al.*, *A General Purpose Transpiler for Fully Homomorphic Encryption*, 2021.

[24] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, ser. SCC '15, Singapore, Republic of Singapore: Association for Computing Machinery, 2015, pp. 13–19.

[25] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, *E3: A Framework for Compiling C++ Programs with Encrypted Operands*, Cryptology ePrint Archive, Paper 2018/1013, 2018.

[26] V. Lab, *CUDA-accelerated Fully Homomorphic Encryption Library*, https://github.com/vernamlab/cuFHE, 2018.

[27] Nucypher, *A GPU implementation of fully homomorphic encryption on torus*, https://github.com/nucypher/nufhe, 2019.