MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers

Siam U. Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, Farinaz Koushanfar
Department of Electrical and Computer Engineering
University of California San Diego
{siamumar, bita, mghasemzadeh, farinaz}@ucsd.edu

ABSTRACT

This paper presents MAXelerator, the first hardware accelerator for privacy-preserving machine learning (ML) on cloud servers. Cloud-based ML is being increasingly employed in various data sensitive scenarios. While it enhances both efficiency and quality of the service, it also raises concern about privacy of the users' data. We create a practical privacy-preserving solution for matrix-based ML on cloud servers. We show that for the majority of the ML applications, the privacy-sensitive computation boils down to either matrix multiplication, which is a repetition of Multiply-Accumulate (MAC) or the MAC itself. We design an FPGA architecture for privacy-preserving MAC to accelerate the ML computation based on the well known Secure Function Evaluation protocol named Yao's Garbled Circuit. MAXelerator demonstrates up to 57× improvement in throughput per core compared to the fastest existing GC framework. We corroborate the effectiveness of the accelerator with real-world case studies in privacy-sensitive scenarios.

CCS CONCEPTS

• **Security and privacy** → *Privacy-preserving protocols*; *Hardware security implementation*;

KEYWORDS

Garbled Circuit, Privacy-preserving computation, Data mining, Secure machine learning, Deep learning

1 INTRODUCTION

Machine Learning (ML) models are increasingly integrated into the cloud services in order to improve the functionality of the underlying application [1–3]. The use of ML models as a cloud service has raised serious questions regarding the information privacy of clients who wants to take advantage of such services. On the one hand, clients do not want to reveal their potentially private input data (e.g., medical records, financial data, or location) to cloud servers. On the other hand, cloud servers should keep their model confidential to preserve the competitive advantage and ensure receiving continuous query requests. As such, it is highly required to devise privacy-preserving frameworks in which, ML models can be executed without disclosing the inputs of each party to one another.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196074

The seminal work [4] by Yao presents the Secure Function Evaluation (SFE) protocol named *Garbled Circuit (GC)* that allows any two-party function to be computed efficiently without revealing their respective inputs. In GC protocol, the underlying function is represented as a Boolean circuit, called a *netlist*. The truth tables of the netlist are encrypted to ensure privacy. In recent years, with the emergence of optimized solutions to the GC protocol, there has been increasing interest in employing GC to ensure the privacy of both the cloud and users in large-scale machine learning and data mining applications [5–7]. While significant algorithmic progress towards efficient privacy-preserving ML has been made, their usage in practical scenarios is still limited by the overhead of SFE operations. Modern ML algorithms including kernel-based data analytics [8, 9] as well as deep learning models [10] rely on iterative matrix multiplication for their execution. Matrix-based ML algorithms are key enablers for devising various contemporary data-driven applications. For instance, the well-known work by Nikolaenko et al. [6] presents a movie recommendation system with private reviews based on GC. Their system takes few hours to operate on a matrix with 10K reviews on a hardware platform with 16 cores. In this paper, we demonstrate how we can effectively reduce this computational time by around 65%. Similar to the work in [6], the bottleneck of the privacy-sensitive computation in the majority of ML applications is matrix multiplication. Real-world applications can be found in various domains such as personal finance (e.g., portfolio analysis [11]), and medical research (e.g., genome analysis [12]). For example, in portfolio analysis [11] the stock correlation data obtained by the financial institution is represented as a matrix and the stock portfolio of the client is represented as a vector and the *risk to return ratio* is the result of a multistage multiplication between these two inputs.

In this work, we present an FPGA accelerator to perform GC-based Multiply-Accumulate (MAC), which is the building block of the matrix multiplication operation. Our approach enables novel and significantly more practicable privacy preserving ML. A number of recent works [13, 14] have presented GC implementation on FPGA. However, their primary focus is on the versatility of the framework rather than computational efficiency. In [13], the underlying netlist is always that of a MIPS processor where the secure function is loaded as a set of instructions. The work in [14], which targets privacy-preserving data mining applications, presents an FPGA overlay architecture[15] where an overlay circuit contains implementations of garbled components (logic gates) upon which the netlist of the secure function is loaded. Both these approaches incur large overhead due to the indirect execution of the GC operation. For example, overlay architectures in general require 40× to 100× more LUTs compared to the conventional design approach [15]. As explained above, in the majority of the ML computations the privacy-sensitive segment of the operation boils down to a MAC.

Therefore we design a concise customized architecture on FPGA to accelerate its GC computation.

Our GC architecture embraces two recent approaches: (1) the TinyGarble [16] framework introduces sequential GC where the same netlist is garbled for multiple rounds with updated encryption keys, (2) the work in [5] perform static analysis on the function (given the control path is independent of the data path) to determine the most optimized netlist to garble in every round. In our design, there are an outer loop and inner loops. The outer loop garbles the netlist of a MAC in every round similar to [16]. The inner loop breaks the operation of the MAC into components such that in every clock cycle we can ensure full utilization of the implemented encryption units. Unlike the conventional GC approach, the underlying netlist is embedded in a finite state machine (FSM) that controls the transfer of the keys between gates. This allows us to employ a parallel architecture for the multiplication operation as we can precisely control the garbling operation in every clock cycle and ensure accurate synchronization among the gates that are garbled in parallel. Unlike a parallelization in software, our approach does not incur any synchronization overhead. Thus we can ensure the minimal idle cycle of the encryption units. As a result, we are able to achieve respectively 57× and 985× improvement in throughput per core compared to [16], which is the fastest software implementation, and the FPGA implementation of [14].
The explicit contributions of this work are the following

- Designing MAXelerator, the first hardware accelerator for privacy-preserving ML on cloud servers. The accelerator is a standalone unit that enables automated integration into reconfigurable cloud architectures.
- Presenting the first GC architecture with precise gate level control per clock cycle. Instead of conventional netlist based GC execution, we design our custom hardware accelerator as an FSM that controls the operation and communication among parallel GC cores, ensuring minimal (highest 2) idle cycles.
- Providing up to 57× improvement in garbling operation compared to the state-of-the-art software GC framework. This translates to the capability of the cloud to support 57× more clients simultaneously.
- Corroborating the effectiveness of the proposed accelerator with real-world case studies in privacy-sensitive scenarios.

2 PRELIMINARIES

2.1 Contemporary ML Practice

Kernel-based machine learning. The optimization objective of many ML applications can be summarized as:

$$Min\ f(x)\ s.t., Ax = y, \tag{1}$$

where y is an observation vector and x is the vector of interest to be found. Examples of the desired minimization function $f(.)$ includes but is not limited to L_1 and L_2 norm of the desired vector x or the geometry of the result. For large matrix sizes, computing the solution of Eq. 1 requires multiple rounds of matrix multiplications:

$$x^{t+1} = x^t - \mu(A^T A x^t - A^T y). \tag{2}$$

μ is a learning rate used in gradient descent based algorithms [8, 17].

Deep learning algorithms. Deep learning is an emerging ML approach that consists of several processing layers stacked on top of one the other. To perform data inference, the raw values of data features are fed into the first layer of the DL network known as the input layer. These raw features are gradually mapped to higher-level abstractions through performing multiple matrix multiplications interleaved by several non-linear operations. The acquired data abstractions are then used to predict the label in the last layer of the DL network. Common DL computations including the convolutional layers can be effectively represented as matrix multiplication as shown in [10, 18].

2.2 Cryptographic Protocols

Oblivious Transfer. Oblivious Transfer (OT)[19] is a cryptographic protocol executed between a sender S and a receiver T, where T selects one from a pair of messages provided by S without revealing her selection. In a 1-out-of-2 OT protocol, (OT_1^2), S holds a pair of messages (m_0, m_1); T holds a selection bit $t \in 0, 1$ and obtains m_t without revealing t to S and learns nothing about m_{1-t}.

Yao's Garbled Circuit. Yao's Garbled Circuit (GC) [4] is a cryptographic protocol where two parties Alice and Bob jointly compute a function $z = \mathcal{F}(a, b)$ on their private inputs a, from Alice and b, from Bob. In the end, the output z is revealed to one or both of them. The function $\mathcal{F}$ is represented as a Boolean circuit, called *netlist*, consisting of 2-input 1-output logic gates. Alice, called the *garbler*, garbles the circuit as follows. She assigns each wire w in the netlist with two k-bit[1] random keys X_w^0 and X_w^1 corresponding to the values 0 and 1, respectively. These keys are called *labels*. For each gate, a garbled truth table is constructed by encrypting the output labels with corresponding input labels. She then sends the garbled tables along with the labels corresponding to her input values to Bob, called the *evaluator*. Bob obtains the labels corresponding to his input values obliviously through OT_1^2. He uses these input labels to evaluate the garbled tables gate by gate. Finally, Alice and Bob share their output maps to learn the output z.

Optimizations of GC. MAXelerator incorporates the following state-of-the art optimizations of the GC protocol

(i) Free XOR [20]. In this method, Alice generates a random $(k-1)$-bit value R which is known only to her. For each wire w, she generates the label X_w^0 and sets $X_w^1 = X_w^0 \oplus (R \parallel 1)$. With this convention, the label for the output wire r of an XOR gates with input wires p, q can be simply computed as $X_r = X_p \oplus X_q$.

(ii) Row Reduction [21]. This optimization reduces the size of the garbled tables for non-XOR gates by 25%. Instead of generating the label for the output wire of a gate randomly, it is computed as a function of the labels of the inputs such that the first entry of the garbled table becomes all 0s and no longer needs to be sent.

(iii) Half Gate [22]. This optimization breaks a non-XOR gate into two half-gates for which one party knows one input. It employs both free XOR and row reduction such that each half-gate can be garbled with a single encryption. As a result, the size of the non-XOR gate truth table is reduced by 50%.

(iv) Garbling With a Fixed-key Block Cipher [23]. This method allows to efficiently garble and evaluate non-XOR gates using fixed-key AES with a unique identifier for each gate.

3 SYSTEM CONFIGURATION

We adopt a cloud server architecture with multiple channels to communicate with the clients as displayed in Figure 1. Along with

[1] k is a security parameter, its value is chosen as 128 in recent works

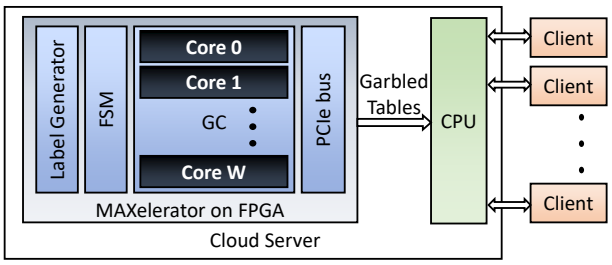

Figure 1: System configuration of MAXelerator framework.

the Central Processing Unit (CPU), the server includes MAXelerator, our FPGA-based accelerator to perform the garbling operation. The MAXelerator creates the garbled tables and sends them to the host CPU that later performs the communication with the client including OT.

MAXelerator consists of the following components (detail description in Section 5):

- Parallel garbling cores to generate the garbled tables. Each core incorporates a GC engine along with a memory block to store the garbled tables.
- Label generator to create the labels necessary for the garbling operation. It includes a hardware Random Number Generator (RNG) to generate the random bit stream.
- An FSM to sync among the garbling cores. The FSM replaces the netlist in the conventional GC. This approach allows us to precisely control the garbling operation customized for the matrix-vector multiplication. Note that the netlist is embedded in the FSM. Therefore, the hardware acceleration is transparent to the evaluator (client) except for the speedup in service.
- A PCIe Bus to transfer the generated garbled tables.

In our proposed setting the cloud server acts as the garbler and the client acts as the evaluator. In general the server possesses the ML model parameters (stored in the form of a set of matrices) and the client holds the input data (in the form of a single vector). Since the evaluator receives his inputs through OT, it is more efficient to have the client, who has less private data, as the evaluator. It is possible to send all the inputs at once through OT extension[24], however, the evaluator may not have enough memory to store all the labels together. With the recent development of sequential GC [16], it is feasible to perform OT every round and store only the labels required for that round; making our approach amenable to memory-constrained clients.

Another motivation behind this setting is that the garbling operation does not require any input from any party. It is only during evaluation that the inputs are required. MAXelerator keeps generating the garbled tables independently and sends them to the host CPU along with the generated labels for the input wires of the netlist. The host in the meantime dynamically updates her model if required, and when requested by the client simply performs the garbling with one of the stored garbled circuits. Note that even if the model does not change, new labels are required for every garbling operation to ensure security.

Security Model. Consistent with most work on GC frameworks, we assume an *honest-but-curious* attack model[16, 23], where the participating parties follow the agreed upon protocol, but may want to deduce more from the information at hand. Our hardware realization does not alter the protocol execution and thus is as secure as any software realization.

Motivation behind hardware acceleration. There are several advantages of an FPGA accelerator over a processor with multiple cores. In a processor, the threads communicate among themselves through shared memory resources. To ensure that the threads do not read stale variables or there are no race conditions we need to create barriers both before and after a thread accessing that memory. The time overhead of the barrier is much higher than the time of generating one garbling table. As a result, parallelizing the GC operation do not result in improvement in timing. Parallelization of garbling operation on GPU is presented in [25, 26], but these works precede the row reduction optimization described in Section 2.2. Therefore, they do not manage the dependency among gates. In FPGA, however, we can precisely control the operation in sync with the clock. Our FSM precisely schedules the garbling operations in the parallel cores to make sure that all the variables (in this case the labels) are written and read in order without the use of a barrier.

4 MAXelerator ARCHITECTURE

The control flow of the MAXelerator architecture comprises two nested loops. The product $Y_{N\times P}$ of two matrices $A_{N\times M}$ $X_{M\times P}$ is

$$Y[i,j] = \sum_{l=0}^{M-1} A[i,l]X[l,j] = \sum_{l=0}^{M-1} a[i]x[j], \tag{3}$$

where a and x are l-th row of A and l-th column of X, respectively. As such, the smallest unit of the matrix multiplication operation consists of a multiplier followed by an accumulator, i.e., a MAC. Following the methodology presented in [16], we design the MAC unit and garble (and evaluate) this unit sequentially for M rounds to compute one element of Y. This forms the outer loop of the control flow. Multiple parallel garbling cores are employed to generate the garbled tables. The number of cores depends on the input bit-width and available resources on the FPGA platform. In the inner loop, we breakdown the operation of the MAC unit such that (1) there is only one non-XOR operation per core per clock cycle, (2) at each cycle, no core is idle due to dependency issues. Note that the cores also contain 1 to 4 XOR gates at every cycle. However, due to the free XOR optimization they do not need costly encryption operations.

We utilize the GC optimized implementation of addition operation with the minimum number of non-XOR gates (one AND gate per input bit) provided in [16]. However, the implementation of the multiplication operation in [16] follows a serial nature that does not allow parallelism. We leverage a tree-based structure for multiplication to maximize parallelism. Figure 2 shows the multiplication operation of two unsigned numbers with bit-width $b = 8$. The operation for signed numbers is discussed later in this section. The bits of x (as well as their corresponding labels in GC operation) are constant over time for one multiplication, and the bits of a (as well as their corresponding labels) are input to the system serially. The addition operations represents one bit full adder where the carry is transferred internally for the next cycle. In the following, we describe the operations of the two segments marked in Figure 2: MUX_ADD and TREE.

4.1 Segment 1: MUX_ADD

The configuration of the parallel GC cores in segment 1 is displayed in Figure 3. One row in the figure represents a GC core on the FPGA,

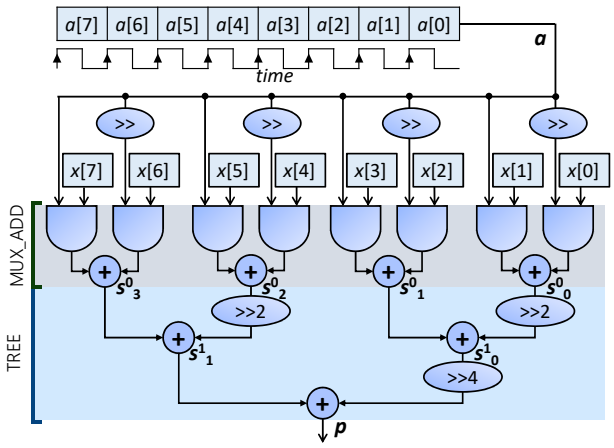

Figure 2: Schematic of the tree-base multiplication.

while one column represents the logic operations performed by that core in every three clock cycles. Henceforth, we refer to every three clock cycles as one *stage*. Each GC core in this segment handles two AND gates and one adder. The adder itself contains one AND and four XOR gates. The logic operations performed in one core per stage is displayed in the inset of Figure 3. The garbling engine of MAXelerator, as described later in Section 5.1, can generate one garbled table per clock cycle. Thus generating the three garbled tables requires three clock cycles, i.e., one stage.

Each core is supplied with a core id m. Core m receives the labels for the two corresponding bits of x: $x[m]$ and $x[m+1]$. These labels remain unchanged for the entire operation. All the cores then receive the labels of two bits of a: $a[n]$ and $a[n+1]$ at each stage n. However, since the garbled table for one gate is generated every clock cycle, each core needs to import only one label per cycle, thus one k-bit input port is sufficient. The label for one bit of a is required for two consecutive stages; thereby at each stage after the first, one label is ported and the other one is shifted internally.

4.2 Segment 2: TREE

At each stage n, a single GC core in segment 1 generates the labels for one bit of the sums: $s_0[n]$, .. $s_{b/2-1}[n]$. At the next stage, these sums are added up in segment 2 according to the tree structure. Since all the cores in segment 1 perform in parallel, the shift operations in Figure 2 translate to delay operations. A d stage delay is realized via d stage k-bit shift register. The number of additions performed in this segment per stage is $b/2+1$. For synchronization with segment 1, the GC cores in this segment is designed to perform three additions per core (three garbled tables, one per each addition). Thus it consists of $\lceil((b/2-1)/3)\rceil$ GC cores.

4.3 Accumulator and Support for Signed Inputs

The final step of the MAC is the accumulator which requires one addition per stage. To support signed inputs, two multiplexer-2's complement pairs are placed at both input and output of the multiplier. Each pair incorporates two AND gates. MAXelerator operates in a pipelined fashion, allowing integration of these nine AND operations into segment 2. This approach results in an increased number of the shift registers. However, it ensures the minimum number of idle cycles for the GC cores. Since, the bottleneck of the resources is the number or LUTs or LUTRAMs, not the number of registers, our approach results in the most optimized design.

Performance Analysis. For bit-width b, MAXelerator requires $b/2+\lceil(b/2+8)/3\rceil$ cores. Thus the maximum number of idle cores is 2. The complete operation takes $b+log(b)+2$ stages. However, since the operations are pipelined, the throughput is 1 MAC per b stages. The final throughput for the multiplication of an $M\times N$ matrix and an $N\times P$ matrix is 1 product per $MNPb$ stages or 1 product per $3MNPb$ cycles.

5 HARDWARE SETTING AND RESULTS

We implement MAXelerator on a Virtex UltraSCALE VCU108 (XCVU095) FPGA. A system with Ubuntu 16.04, 128 GB memory, and Intel Xeon E5-2600 CPU @ 2.2GHz is employed as the host CPU, as well the software platform for performance comparison. We leverage PCIe library provided by [27] to interconnect the host and FPGA platforms.

5.1 GC Engine

Each GC core incorporates one GC engine that generates one garbled table per clock cycle. It adopts all the optimizations described in Section 2.2 The GC engine takes the labels for the two input wires of the AND gate as its input and outputs the corresponding garbled tables. Our implementation involves only two gates: AND and XOR. Due to free XOR optimization, XOR gates just require XORing the two input labels and are handle outside while the GC engine is designed to generate garbled tables only for the AND gates. This approach ensures that there is no mismatch in the timing for executing different gates as in [13] and therefore no stalling due to dependency issues. According to the methodology presented in [23], the encryption is performed by fixed-key block cipher instantiated with AES. We employ a single stage AES implementation. The s-boxes inside the AES algorithm are implemented efficiently by utilizing the LUTRAMs on the Virtex UltraSCALE FPGA. The unique gate identifier T is generated by concatenating i, j (see Eq. 3), core id, stage index and and gate id (see Figure 3).

The on-chip memory on the FPGA is divided into blocks with one input port per block and one output port for the entire memory. The output port is used by the PCIe Bus to transfer the generated input labels and garbled tables to the general purpose processor hosting the FPGA. Since each core has its own block in the memory with an individual input port, logically it can be visualized as each core having its own memory block.

5.2 Label Generator

To generate the wire labels for GC we implement on-chip hardware Random Number Generators (RNG). We adopt the Ring Oscillator (RO) based RNG suggested by Wold et al. in [28]. One RNG XORs the output of 16 ROs each containing 3 inverters. The entropy of the implemented RNG on our evaluation platform is thoroughly evaluated by NIST battery of randomness tests. In the worst-case scenario, the GC accelerator requires $k\times(b/2)$ random bits/cycle. However, for a large portion of the operation, it requires only k bits/cycle on average. The label generator incorporates $k\times(b/2)$ RNGs such that it can support the worst-case setting. The FSM that synchronize the garbling operation fully or partially turns off the operation of the RNGs to conserve energy, when possible.

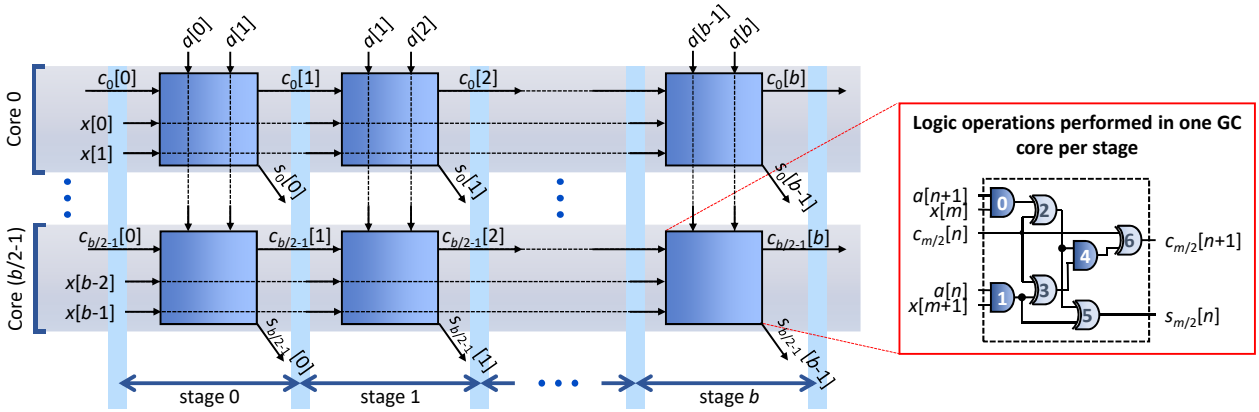

Figure 3: The high-level configuration and functionality of the parallel GC cores in segment 1: MUX_ADD

5.3 Resource Usage

The FPGA resource usage of one MAC unit is shown in Table 1 for different bit-widths b. It can be seen from the table that the underlying resource utilization of our design increases linearly with b. The maximum clock frequency supported by this implementation is 200MHz on the Virtex UltraSCALE. We do not compare the resource usage with the prior-art GC implementation on FPGA [14] for two reasons: (i) [14] being a generic GC implementation, it is difficult to estimate the resource it would require only to perform the MAC operation in similar number of clock cycles as this work, (ii) it employs SHA-1 for encryption (the most resource consuming part of the implementation), while we employ AES. SHA-1 is not considered secure anymore and all the current GC implementations in both software and hardware employ AES.

Table 1: Resource usage of one MAC unit

Bit-width (b)	8	16	32
LUT	2.95E+04	5.91E+04	1.11E+05
LUTRAM	1.28E+02	3.84E+02	6.40E+02
Flip-Flop	2.44E+04	4.88E+04	8.40E+04

5.4 Comparison with the Prior-art

To the best of our knowledge, MAXelerator is the first custom FPGA implementation of privacy-preserving MAC. Table 2 compares the physical performance of MAXelerator against the fastest available software GC framework TinyGarble [16] and the FPGA GC solution presented in [14]. Both MAXelerator and [14] employ parallel GC cores to accelerate the operation. In our work, the maximum number of parallel cores depends on the available resources in the FPGA while in [14] it depends on the latency of garbling one AND gate and available BRAMs. Taking all these into account we believe that reporting the overall throughput would be ambiguous and somewhat unfair to the software framework [16]. Therefore, we report the performances of all the frameworks per core.

As shown Table 2 MAXelerator accelerates the garbling operation by up to 57× compared to [16] and at least 985× compared to [14]. Another recent work, GarbledCPU [13] do not report evaluation results for multiplication and addition. However, they report 2× improvement in throughput compared to JustGarbled [23] (which is the back-end of [16]) on an Intel Core i7-2600 CPU @ 3.4GHz. We estimate at least 37× improvement over [13] in throughput per core (this work does not attempt parallelization). The throughput of [16] will go down while garbling a complete netlist due to pipeline stalls caused by dependency issues.

To be fair, we should state that a major factor behind the lower throughput of [14, 16] is due to their focus on general purpose GC computing while MAXelerator is custom made for performing matrix multiplication only. However, the large improvement in throughput establishes the practicality of the custom solution. We bolster this further through case studies in the next section.

6 CASE STUDIES

In this section, we analyze a number of well-known ML applications to asses the speedup provided by the custom FPGA realization of a GC based MAC. We assume a 32 bit fixed point system with 24 cores on MAXelerator. Note that the throughput can be increased linearly by adding more GC cores to the FPGA. For example, 25 times more GC cores can fit in our current implementation platform.

Recommendation System: The work in [6] yields an efficient implementation of matrix factorization with application in movie recommender system which has been widely adopted by many other works such as [29, 30]. As authors mentioned in [6], more than 2/3 of the execution time is spent on vector multiplication for gradient computations. The complexity of the proposed matrix factorization is $O(Mlog^2M)$ where M is the total number of ratings while the complexity of the pertinent MAC operations in each operation is $O(Sd)$ where S is summation of total number of ratings and total number of movies, and d is the dimension of user/item profile. On the MovieLens dataset each iteration of [6] takes 2.9hr. Incorporating our hardware accelerated MAC into the approach of [6] significantly reduces the gradient computation time, decreasing the total runtime per iteration from 2.9hr to 1hr (69% improvement).

Ridge Regression: This method is used to find the best-fit linear curve through input data points. The work in [7] combines both homomorphic encryption and Yao garbled circuits to efficiently perform private ridge regression. The approach proposed in [7] has $O(d^3)$ MACs, $O(d)$ square roots, and $O(d^2)$ divisions in the first phase and $O(d^2)$ MAC operations in the second phase. As such, accelerating the MAC operations would significantly improve the runtime as shown in Table 3 for selected datasets of [7]. n and d are number of samples and feature size respectively.

Table 2: Throughput Comparison of MAXelerator with state-of-the-art GC frameworks

	TinyGarble [16] on CPU			FPGA Overlay Architecture [14]			MAXelerator on FPGA		
Bit-width	8	16	32	8	16	32	8	16	32
Clock Cycle per MAC	1.44E+05	5.45E+05	2.24E+06	4.40E+03	1.20E+04	3.60E+04	24	48	96
Time per MAC (μs)	42.29	160.35	657.65	22.00	60.00	180.00	0.12	0.24	0.48
Throughput (MAC per sec)	2.36E+04	6.24E+03	1.52E+03	4.55E+04	1.67E+04	5.56E+03	8.33E+06	4.17E+06	2.08E+06
No of cores	1	1	1	43	43	43	8	14	24
Throughput per core (MAC per sec)	2.36E+04	6.24E+03	1.52E+03	1.06E+03	3.88E+02	1.29E+02	1.04E+06	2.98E+05	8.68E+04
× Throughput of MAXelerator per core	1/44	1/48	1/57	1/985	1/768	1/672	-	-	-

*Interpolated from the results provided in [14] for 8, 32 and 64 bits.

Table 3: Ridge Regression Runtime Improvement

Name	n	d	Time (s) ([7])	Time (s) (Ours)	Runtime Impr.
communities11.IV	2215	20	314	7.8	39.8 ×
automobile.I	205	14	100	3.5	28.4 ×
forestFires	517	12	46	1.8	24.5 ×
winequality-red	1599	11	39	1.7	22.6 ×
autompg	398	9	21	1.1	18.7 ×
concreteStrength	1030	8	17	1.0	16.8 ×

Portfolio Analysis: To calculate the current risk to return ratio based on the stock portfolio of the investor, the client stock weight vector w (which contains relative weight of stocks in the investor's portfolio) and the financial institution stock covariance matrix cov (which is the result of financial institution's research on the market) are required. The risk to return ratio is then obtained by performing $w \times cov \times w'$ where w' is the transpose of w [11]. In [31], the authors reported 20μs to perform 252 rounds of risk to return analysis for a portfolio of size 2 on Nvidia-k80 GPU. According to our evaluation, the same computation with privacy-preserving would take 1.33 seconds using TinyGarble and 15.23ms using MAXelerator.

In the above analysis we assumed that the cloud server has sufficient communication channels. However, after certain threshold, communication capability of the server may become the bottleneck of the operation.

7 CONCLUSION

We present MAXelerator, an efficient FPGA implementation of GC based MAC to accelerate privacy-preserving machine learning on cloud servers. MAXelerator achieves up to 57× improvement in throughput per core compared to the fastest GC framework. Our acceleration focus is on matrix multiplication which is the most costly component in several key ML applications. Acceleration of this process can bring down the operational time in the privacy-sensitive scenario to practical limits, as verified by our case studies.

Acknowledgments. This work was supported in parts by the ONR (N00014-17-1-2500), NSF/SRC (CNS-1619261) and NSF (CNS-1649423) grants.

REFERENCES

[1] N. Jones *et al.*, "The learning machines," *Nature*, vol. 505, no. 7482, 2014.
[2] J. Kirk, "Ibm join forces to build a brain-like computer," http://www.pcworld.com/article/2051501/, 2016.
[3] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Deep3: Leveraging three levels of parallelism for efficient deep learning," in *DAC*. ACM, 2017.
[4] A. C.-C. Yao, "How to generate and exchange secrets," in *Annual Symposium on Foundations of Computer Science*. IEEE, 1986.
[5] X. Wang, S. D. Gordon, A. McIntosh, and J. Katz, "Secure computation of mips machine code," in *ESORICS*. Springer, 2016.
[6] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *Conference on Computer & communications security*. ACM, 2013.
[7] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *Symposium on S & P*. IEEE, 2013.
[8] M. Journée, Y. Nesterov, P. Richtárik, and R. Sepulchre, "Generalized power method for sparse principal component analysis," *Journal of Machine Learning Research*, vol. 11, no. Feb, 2010.
[9] C. Fowlkes, S. Belongie, F. Chung, and J. Malik, "Spectral grouping using the nystrom method," *Trans. on pattern analysis and machine intelligence*, 2004.
[10] L. Deng, D. Yu *et al.*, "Deep learning: methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, 2014.
[11] G. Connor, L. R. Goldberg, and R. A. Korajczyk, *Portfolio risk analysis*. Princeton University Press, 2010.
[12] H. Krcmar, R. Reussner, and B. Rumpe, *Trusted cloud computing*. Springer, 2014.
[13] E. M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar, "Garbledcpu: a mips processor for secure computation in hardware," in *Design Automation Conference*. ACM, 2016.
[14] X. Fang, S. Ioannidis, and M. Leeser, "Secure function evaluation using an fpga overlay architecture." in *FPGA*, 2017.
[15] A. Brant and G. G. Lemieux, "Zuma: An open fpga overlay architecture," in *Field-Programmable Custom Computing Machines*. IEEE, 2012.
[16] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *Symposium on S&P*. IEEE, 2015.
[17] B. D. Rouhani, A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Automated real-time analysis of streaming big and dense data on reconfigurable platforms," *TRETS*, vol. 10, no. 1, 2016.
[18] Z. Song, Z. Liu, C. Wang, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," *arXiv preprint arXiv:1709.07776*, 2017.
[19] M. Naor and B. Pinkas, "Computationally secure oblivious transfer," in *Journal of Cryptology*, vol. 18, no. 1. Springer, 2005.
[20] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *Automata, Languages and Programming*. Springer, 2008.
[21] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *Conference on Electronic commerce*. ACM, 1999.
[22] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates." 2015.
[23] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *Symposium on S & P*. IEEE, 2013.
[24] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently." in *Crypto*, vol. 2729. Springer, 2003.
[25] S. Pu, P. Duan, and J.-C. Liu, "Fastplay-a parallelization model and implementation of smc on cuda based gpu cluster architecture." *IACR Cryptology ePrint Archive*, 2011.
[26] N. Husted, S. Myers, A. Shelat, and P. Grubbs, "Gpu and cpu parallelization of honest-but-curious secure two-party computation," in *Computer Security Applications Conference*. ACM, 2013.
[27] XILLYBUS, "http://xillybus.com/," 2017.
[28] K. Wold and C. H. Tan, "Analysis and enhancement of random number generator in fpga based on oscillator rings," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.
[29] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic protocol selection in secure two-party computations," in *International Conference on Applied Cryptography and Network Security*. Springer, 2014.
[30] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi, "Scoram: oblivious ram for secure computation," in *CCS*. ACM, 2014.
[31] J. A. Varela and N. Wehn, "Near real-time risk simulation of complex portfolios on heterogeneous computing systems with opencl," in *International Workshop on OpenCL*. ACM, 2017.