2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)

Accelerating Large Garbled Circuits on an FPGA-enabled Cloud

Miriam Leeser
Dept of ECE
Northeastern University
Boston, MA
mel@coe.neu.edu

Mehmet Gungor
Dept of ECE
Northeastern University
Boston, MA
gungor.m@husky.neu.edu

Kai Huang
Dept of ECE
Northeastern University
Boston, MA
huang.kai1@husky.neu.edu

Stratis Ioannidis
Dept of ECE
Northeastern University
Boston, MA
ioannidis@ece.neu.edu

***Abstract*—Garbled Circuits (GC) is a technique for ensuring the privacy of inputs from users and is particularly well suited for FPGA implementations in the cloud where data analytics is frequently run. Secure Function Evaluation, such as that enabled by GC, is orders of magnitude slower than processing in the clear. We present our best implementation of GC on Amazon Web Services (AWS) that implements garbling on Amazon's FPGA enabled F1 instances. In this paper we present the largest problems garbled to date on FPGA instances, which includes problems that are represented by over four million gates. Our implementation speeds up garbling 20 times over software over a range of different circuit sizes.**

***Index Terms*—privacy, garbled circuits, high performance computing, FPGAs**

I. Introduction

Privacy has become an increasing concern among users of computer systems and services. Secure Function Evaluation (SFE) is an approach then ensures users data privacy. However, such assurances require a significant increase in processing requirements and latency. The two main techniques used for SFE are homomorphic encryption and garbled circuits (GC). This research focuses on accelerating GC, which is an excellent match for FPGAs in the data center. In particular, this research targets FPGAs in the data center, and processing large problems.

Previous approaches to accelerating GC with FPGAs have focused on embedded systems and small problems, especially those where all intermediate data fits in on-chip memory [1], [2]. Our recent research accelerates FPGAs in the cloud, specifically with AWS F1 instances [3]. In a recent paper [4], we describe our first results accelerating GC in the cloud. In this paper, we discuss improvements to those initial results and how we handle larger circuits.

One of the distinguishing features of this research is to make use of large data sets that do not entirely fit in on-chip memory and that require random access. To improve on our previously published research, we introduce a hybrid design that makes use of both on-chip and off-chip memory. We use the on-chip memory as a managed cache that keeps as much data as possible close to the processing for acceleration. The memory management technique presented is also applicable to other big data problems that make use of FPGAs.

In this paper we describe approaches and experiments to improve on previous implementations in order to obtain improved speedup as well as support larger examples. Specifically, the main contribution of this paper is to use the memory available on an FPGA more efficiently in order to support large data sets on FPGA instances and improve the speed up obtained from running applications on FPGAs in the cloud.

The rest of this paper is organized as follows. Section II provides background on garbled circuits and AWS as well as presenting related work. In section III we discuss our implementation of the garbler and highlight the improvements presented here over previously published results. Section IV presents experimental results. Finally, we conclude and present plans for future work.

II. Background

A. Garbled Circuits

Our research accelerates Secure Function Evaluation (SFE), specifically Garbled Circuits (GC), using FPGAs. In this model there are two or more users with data which they wish to keep private, and a function to be evaluated over that data. All parties know the function being evaluated and learn the outcome of the evaluation, but users do not reveal their data. The threat model we follow is "honest but curious", where an adversary follows the protocol as specified, but tries to learn as much as possible. A canonical problem exemplifying SFE is the "Millionaires' Problem:" two millionaires wish to know who is worth more without revealing their personal worth to each other.

Garbled circuits were initially introduced by Yao [5] for two users and have been extended to multiple users. They rely on cryptographic primitives. In the variant we study here (adapted from [6], [7]), Yao's protocol runs between (a) a set of private input owners, (b) an Evaluator, who wishes to evaluate a function over the private inputs, and (c) a third party called the Garbler, that facilities and enables the secure computation.

Garbled Circuits work for any problem that can be expressed as a Boolean circuit. In our and many other implementations, this function is represented as a circuit made up of AND and

This research is funded by NSF under award SaTC 1717213.

978-1-7281-5999-7/19/$31.00 ©2019 IEEE
DOI 10.1109/H2RC49586.2019.00008

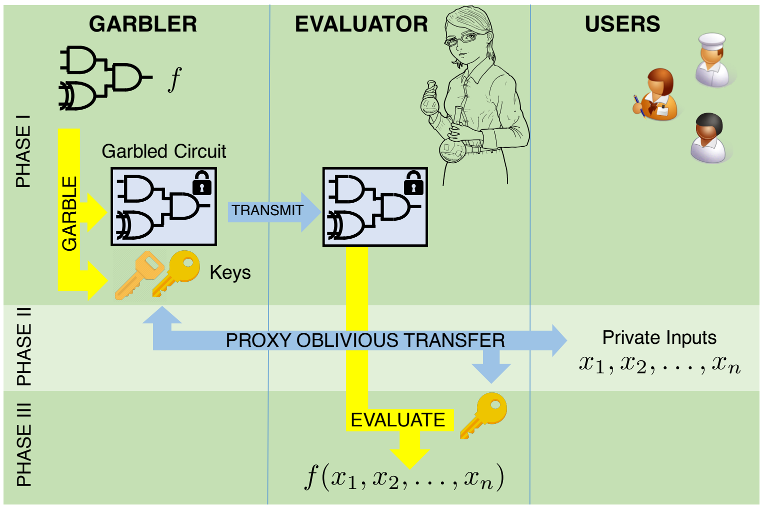
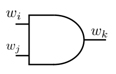


Fig. 1. Yao's Protocol Phases of Operation

XOR gates.[1] The Evaluator wishes to evaluate a function f, represented as a Boolean circuit of AND and XOR gates, over private user inputs $x_1, x_2, \ldots, x_n$. We break the problem into three phases, as shown in Fig. 1. In Phase I, the Garbler "garbles" each gate of the circuit, outputting (a) a "garbled circuit," namely, the garbled representation of every gate in the circuit representing f, and (b) a set of keys, each corresponding to a possible value in the string representing the inputs $x_1, \ldots, x_n$. These values are shared with the Evaluator. In Phase II, through proxy oblivious transfer [8], the Evaluator learns the keys corresponding to the true user inputs. In the final phase, the Evaluator uses the keys as input to the garbled circuit to evaluate the circuit, un-garbling the gates. At the conclusion of Phase III, the Evaluator learns $f(x_1, \ldots, x_n)$.

1) Garbling Phase: A function to be evaluated is represented as a Boolean circuit consisting of AND and XOR gates. In the garbling phase, each of these gates is garbled as described in this section. Each gate is associated with three wires: two input wires and one output wire. At the beginning of the garbling phase, the Garbler associates two random strings, $k_{w_i}^0$ and $k_{w_i}^1$, with each wire w_i in the circuit. Intuitively, each $k_{w_i}^b$ is an encoding of the bit-value $b \in \{0, 1\}$ that the wire w_i can take.



b_i	b_j	$g(b_i, b_j)$	Garbled value
0	0	0	$Enc_{(k_{w_i}^0, k_{w_j}^0, g)}(k_{w_k}^0)$
0	1	0	$Enc_{(k_{w_i}^0, k_{w_j}^1, g)}(k_{w_k}^0)$
1	0	0	$Enc_{(k_{w_i}^1, k_{w_j}^0, g)}(k_{w_k}^0)$
1	1	1	$Enc_{(k_{w_i}^1, k_{w_j}^1, g)}(k_{w_k}^1)$

Fig. 2. A Garbled AND Gate

We describe here how to garble an AND gate. The same principles can be applied to garble an XOR gate, using its respective truth table. We note however that, in practice, XOR gates are handled via the Free XOR optimization [9], discussed in Section II-A3. A garbled AND gate is shown in Fig. 2. For each AND gate g, with input wires (w_i, w_j) and output wire w_k, the Garbler computes the following four ciphertexts, one for each pair of values $b_i, b_j \in \{0, 1\}$:

$$Enc_{(k_{w_i}^{b_i}, k_{w_j}^{b_j}, g)}(k_{w_k}^{g(b_i,b_j)}) = SHA(k_{w_i}^{b_i} \| k_{w_j}^{b_j} \| g) \oplus k_{w_k}^{g(b_i,b_j)} \tag{1}$$

Here SHA represents the hash function, $\|$ indicates concatenation, g is an identifier for the gate, and $\oplus$ is the XOR operation. Note that each value k on a wire is implemented with 80 bits in our implementation. The "garbled" gate is then represented by a random permutation of these four ciphertexts. Observe that, given the pair of keys $(k_{w_i}^0, k_{w_j}^1)$ it is possible to successfully recover the key $k_{w_k}^1$ by decrypting $c = Enc_{(k_{w_i}^0, k_{w_j}^1, g)}(k_{w_k}^1)$ through:[2]

$$Dec_{(k_{w_i}^0, k_{w_j}^1, g)}(c) = SHA(k_{w_i}^{b_i} \| k_{w_j}^{b_j} \| g) \oplus c \tag{2}$$

On the other hand, the other output wire key, namely $K_{w_k}^0$, cannot be recovered. More generally, it is worth noting that the knowledge of (a) the ciphertexts, and (b) keys $(k_{w_i}^{b_i}, k_{w_j}^{b_j})$ for some inputs b_i and b_j yields *only* the value of key $k_{w_k}^{g(b_i,b_j)}$; no other input or output keys of gate g can be recovered. Any Boolean function can be garbled in this manner, by first representing it with ANDs and XORs, and then garbling each such gate.

Our implementation makes use of SHA cores for encryption and decryption. Although SHA has been found to be vulnerable in general, since we re-encrypt for every new input value, this implementation is secure. We compare our results, both

[1]Recall that AND and XOR gates form a complete basis for Boolean circuits.

[2]Note that the above encryption scheme is *symmetric* as Enc and Dec are the same function.

values and run times to FlexSC [10] which also makes use of SHA. We plan to update the cores to AES. As the core speed and area is not currently a bottleneck in our design, we do not expect this to effect the result presented here.

2) Evaluation Phase: The output of the garbling process is (a) the garbled gates, each comprising a random permutation of the four ciphertexts representing each gate, and (b) the keys $(k^0_{w_i}, k^1_{w_i})$ for every wire w_i in the circuit. At the conclusion of the first phase, the Garbler sends this information for all garbled gates to the Evaluator. It also provides the correspondence between the garbled value and the real bit-value for the circuit-output wires (the outcome of the computation): if w_k is a circuit-output wire, the pairs $(k^0_{w_k}, 0)$ and $(k^1_{w_k}, 1)$ are given to the Evaluator. To transfer the garbled values of the input wires, the Garbler engages in a proxy oblivious transfer with the Evaluator and the users, so that the Evaluator obliviously obtains the garbled-circuit input value keys $k^b_{w_i}$ corresponding to the actual bit b of input wire w_i.

Having the garbled inputs, the Evaluator can "evaluate" each gate, by decrypting each ciphertext of a gate in the first layer of the circuit by applying equation (2): only one of these decryptions will succeed,[3] revealing the key corresponding to the output of this gate. Each output key revealed can subsequently be used to evaluate any gate that uses it as an input. Using the table mapping these keys to bits, the Evaluator can learn the final output.

3) Optimization: Several improvements over the original Yao's protocol have been proposed, that lead to both computational and communication cost reductions. These include point-and-permute [11], row reduction [12], and Free-XOR [9] optimizations, all of which we implement in our design. Free-XOR in particular significantly reduces the computational cost of garbled XOR gates: XOR gates do not need to be encrypted and decrypted, as the XOR output wire key is computed through an XOR of the corresponding input keys. In addition, the free-XOR optimization fully eliminates communication between the Garbler and the Evaluator for XOR gates: no ciphertexts need to be communicated for these gates. Our implementation takes advantage of all of these optimizations; as a result, the circuit for computing garbled AND gates differs slightly from the garbling algorithm outlined above.

B. AWS

Amazon Web Service (AWS) provides cloud computing, data storage and other data-relevant services for enterprise development, personal use and academic research. Specifically, AWS has f1 type instances that use FPGAs from Xilinx to enable delivery of custom hardware acceleration [3]. We use the f1.2xlarge with Virtex Ultrascale+ ECVU9p FPGAs. The FPGA board includes 4x16 GB external DDR memory.

AWS provides several different ways to program their FPGAs. We use the AWS-FPGA Hardware Development Kit (HDK) which provides development support and runtime libraries for their FPGA instances. The SDK supports OpenCL development; however, since we are developing an overlay architecture it is not a good match for our design.

AWS-FPGA hardware infrastructure connects the FPGA board, which includes external DDR memory, to the host mother board through the PCIe bus. The interconnect with AXI protocol in the hardware design enables data movement between host memory, FPGA on-chip memory and external DDR memory on the FPGA board. Additionally, the software runtime library provides API interfaces to transfer chunks of data to DDR memory and interfaces to access on-chip memory in the FPGA.

C. Related Work

Acceleration of garbled circuits on FPGAs is an active area of research. TinyGarble [13] uses techniques from hardware design to implement GCs as sequential circuits and then optimizes these designs. The circuits can be optimized to reduce the non-XOR operations using traditional high-level synthesis tools. The resulting designs are customized for each problem; thus for each new problem a new bitstream must be generated, hence it is not practical for large designs in a data center setting. We implement as many garbled AND gates as we can keep busy at the same time, and implement garbled circuits directly on top of an efficient overlay, which eliminates the need to recompile the hardware for every new user problem. In MAXelerator [14] the authors implement an efficient garbling of matrix multiplication in FPGAs. With RedCrypt [15], the same authors use their design to garble larger examples. Their approach differs from ours in that the FPGA is used as a streaming accelerator and intermediate results are not stored in memory on the FPGA. Hence, their design requires a very high bandwidth for host to FPGA communications. The results presented do not include the time required for this communication. Also, they generate random values local to the FPGA, while we generate them on the host. There are two advantages for using the host for this. One is the ability to compare software and hardware results for validation. The second is that random input strings must be sent to the evaluator so the host requires this information.

In previous work, we use an FPGA with a local host for accelerating general garbled circuit problems and demonstrate orders of magnitude improvement over the software version [1], [16], [17]. The acceleration is achieved via an FPGA overlay architecture, hybrid memory hierarchy, efficient data manipulation and fine grained communication patterns between the host and FPGA. This previous work targets one FPGA with a Stratix V. It only implements the garbler and not the evaluator.

There are several FPGA projects that target AWS f1 instances. In [18], the CAOS framework is extended to integrate with SDAccel running on AWS instances to improve performance. In [19], the authors developed an FPGA-based ultrasonic propagation imaging system to process real-time ultrasonic signals. Firesim [20] builds a cycle-exact simulation platform on large-scale clusters integrated with FPGA accel-

[3]This can be detected, e.g., by appending a prefix of zeros to each key $k^b_{w_k}$, and checking if this prefix is present upon decryption.

erators to simulate behaviors of data movement among CPU, caches, DRAM and network switches.

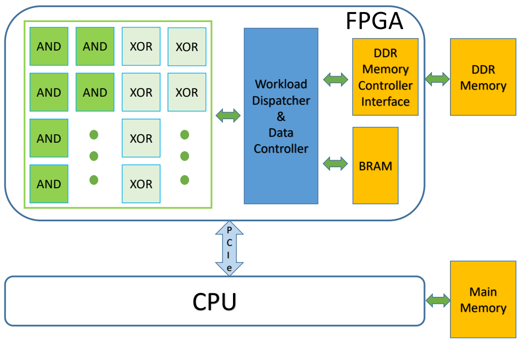

FPGA
AND
XOR
Workload Dispatcher & Data Controller
DDR Memory Controller Interface
DDR Memory
BRAM
PCIe
CPU
Main Memory

Fig. 3. Overlay Architecture implemented on FPGA

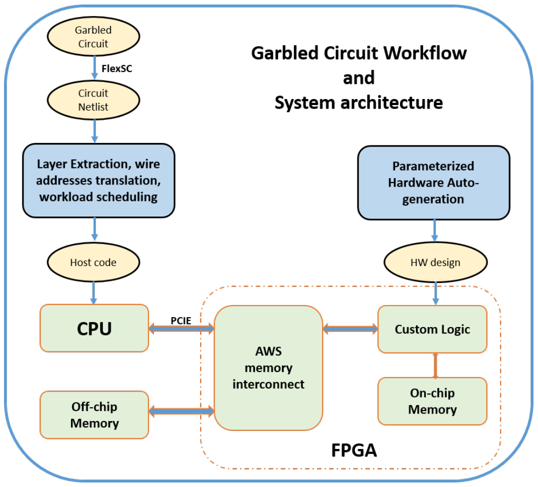

Garbled Circuit
FlexSC
Circuit Netlist
Garbled Circuit Workflow and System architecture
Layer Extraction, wire addresses translation, workload scheduling
Parameterized Hardware Auto-generation
Host code
HW design
CPU
PCIE
AWS memory interconnect
Custom Logic
On-chip Memory
Off-chip Memory
FPGA

Fig. 4. Workflow of our design

III. Improvements to Support GC in the Datacenter

In a recent paper we presented initial experiments for using AWS F1 instances for accelerating garbled circuits [4]. This previous paper looks at the entire system of garbling, transferring data and evaluating a function using garbling. In our system, the garbler is implemented with an F1 instance while the evaluator is implemented on an independent node in software. In this paper we focus on the garbler implementation and discuss improvements to the base design specifically for supporting large circuits with large memory footprints.

Our design makes use of a coarse grained overlay architecture as shown in Fig. 3. This architecture implements a sea of garbling AND and garbling XOR gates on the FPGA fabric. For a particular set of experiments the number of implemented gates is fixed; we discuss this in the results section of this paper. Individual problems are mapped onto the implemented architecture. Note that the design supports the use of both BRAM and DDR memory, an important feature for this paper. This is an overlay architecture because the implementation is fixed and many different user problems can be mapped to it. Thus the FPGA does not need to be redesigned for each problem instance. Because of its coarse grained nature, there is very little overhead incurred.

Fig. 4 gives an overview of the software and hardware used to garble a problem instance. We use FlexSC [10] as a front end to generate the circuit netlist, which is shared between the garbler and evaluator. We do preprocessing on the host to generate layers (garbling is done in breadth-first order) and assign wires to memory locations on the FPGA. The hardware design on the FPGA is generated once for a set of experiments. We also use FlexSC to validate our designs.

The flow of the garbler works as follows. Two types of data are generated on the host PC and communicated to the FPGA over PCIe: keys that correspond to the inputs and addresses that correspond to wire locations in the garbled circuit. Intermediate keys are generated on the FPGA. These are required for the garbling process, but do not need to be communicated to the evaluator. The evaluator does require the garble tables; one such table is generated for each garbled AND gate in the circuit.

The circuits we are interested in garbling can grow to well over a million gates. In the design reported in [4] all intermediate keys are stored in DDR memory, which is off chip, and has a latency of between 30 and 55 clock cycles for each access. Note that memory access of these intermediate keys is random so we cannot take advantage of burst mode to reduce latency. In the hybrid design reported in this paper, we make use of both off chip DDR memory and on-chip Block RAM (BRAM). Accessing BRAM on the FPGA is 30 times or more faster than accessing DDR memory. In the hybrid design, initial keys are stored in DDR memory, while intermediate results are stored in BRAM as long as they fit. Using preprocessing on the host, we keep track of lifetimes of intermediate values. When we pre-process the netlist, we can have a 'reference count' for each wire and we will reduce the count in execution. When the count reaches zero, it means this BRAM location can be reused. When an intermediate value is no longer needed, we set a flag to show that the BRAM location can be over-written and reused. All our results for circuits whose intermediate designs do not completely fit into memory reuse BRAM locations. In our previous results, writing intermediate results to DDR took approximately 30% of the total runtime. This has been significantly reduced in the results reported here.

We use DDR for initial input keys, which are generated on the host and sent directly to DDR. As noted above, these input keys are also needed by the evaluator, so it makes sense to generate them on the host. We use DDR for the garble tables generated, which also need to be transferred to the evaluator. Intermediate values generated during garbling are not transmitted. We use BRAM for when these values fit and DDR when the amount of BRAM on the FPGA is not sufficient. On AWS hardware, a DDR Read takes slightly more than 50 cycles, while a DDR write takes more than 30 clock cycles. In most cases, since accesses are random, the efficiencies of burst mode access cannot be taken advantage of. In comparison, a BRAM read can be achieved in 1 clock cycle.

Accessing off chip memory in random order can incur long latencies. We minimize memory accesses by storing intermediate values generated during garbling in BRAM. BRAM is organized as 10000 512 bit wide locations. We continue to investigate different organizations of BRAM to maximize use. We assign values to BRAM using a greedy algorithm, and continue to store data in BRAM as locations are available. We reuse locations when an intermediate value is no longer needed. If currently all BRAM locations are occupied, the data will be stored to DDR. Others are investigating algorithms for random access of BRAM [21]. We plan to investigate this approach further.

We generate the FPGA design from Python, which allows us to easily change the number of Garbled AND (gAND) and Garbled XOR (gXOR) cores implemented in hardware. We show results for two different designs: four gAND and four gXOR cores, and eight gAND and eight gXOR cores. Note that these values can easily be changed and that the number of AND cores does not need to be the same as the number of XOR cores.

IV. Results

A. Experimental Setup

We run all our designs on one Amazon F1 instance with attached processors. We use the f1.2xlarge with Virtex Ultrascale+ ECVU9p FPGAs. The FPGA board includes 4x16 GB external DDR memory. DDR memory, as well as registers on the FPGA can be directly written by the host using the AWS HDK and SDK frameworks.

Input values and address locations are stored in FPGA registers by the host processor. The output of garbling is the garble tables which are stored in DDR memory by the FPGA and transferred back to the host. Times are measured from the time between setting up registers for all gates and writing all garble tables for all gates in the netlist to DDR. Therefore it represents end-to-end run times for FPGA processing of garbling of the circuit.

B. Experimental Results

In this paper we are focusing on accelerating time for garbling, which is longer than the time required for evaluation [4]. We show run times for FlexSC to garble circuits in software, run times for running on the FPGA using only DDR memory to store intermediate results, and the hybrid design introduced in this paper that uses both BRAM and DDR memory. The circuit examples are all examples of different sizes of matrix-matrix multiply (labeled MM). Matrix multiplication is useful to see how results scale for different sizes of problem. Our approach is general and can handle any problem that can be represented as a Boolean circuit. We vary the size of the matrices being multiplied and the number of bits to represent data in order to generate different sizes of problem. The problems presented here range from 15,000 to over four million gates.

In Table I, we compare run times across these problem sizes for using DDR memory only and the hybrid design that uses both DDR and BRAM memory. We also show results comparing two different overlays, one with four garbled AND and four garbled OR circuits and the other with eight garbled AND and eight garbled OR circuits. Note that an arbitrary number of garbled gates can be generated for a specific overlay, but once it has been designed the number of gates is fixed. Also, we chose the same number of gAND and gXOR gates in these experiments, but this is not required.

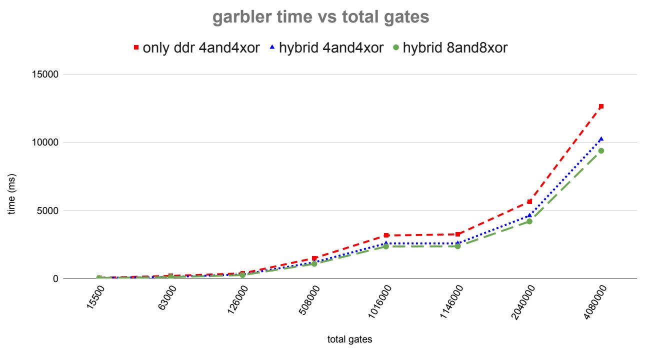


Fig. 5. Time vs total number of gates on the FPGA

Results in Table I show that there is a consistent improvement in using BRAM compared to the designs that use DDR alone. The speedup is greatest for the smaller designs (5×5 Matrix Multiply) where all intermediate results can be stored in Block RAM. Even in cases where both types of memory are required, we consistently get about a 25% increase in performance by using local memory. Table I also shows that the overlay containing 8 gAND and 8 gXOR gates consistently outperforms the overlay with fewer cores instantiated by about an additional 10% improvement. The number of cores and organization of the memory hierarchy are linked. We continue to experiment with different numbers of garbled ANDs and garbled XOR gates as well as different organizations of BRAM to determine the optimal architecture. Figure 5 summarizes the different FPGA versions as a function of size of the circuit. the hybrid memory design consistently improves over the DDR only design, and the increased number of cores provides the best results across all sizes of circuit.

Table II shows the results for the same examples with a comparison between software run times, and the best performing design from Table I, i.e., the hybrid memory design with 8 gAND and 8 gXOR cores. Once again, the speedup across all different circuit sizes is pretty consistent at about 20 times improvement. The speedup diminishes as the circuit gets larger. This is likely due to the fact that the percent of DDR fetches increases for larger circuit sizes. We are investigating the optimal number of cores as well as the best way to assign intermediate values to BRAM vs. DDR RAM.

TABLE I
GARBLER TIMING DDR VS HYBRID MEMORY DESIGN. ALL UNITS ARE MS.

	total gates	4 AND 4 XOR DDR	4 AND 4 XOR hybrid	speedup	8 AND 8 XOR	speedup (over 4 and 4)
4bit 5×5 MM	15500	45.48	29.47	1.54	26.42	1.12
8bit 5×5 MM	63000	184.23	111.74	1.65	96.61	1.16
4bit 10×10 MM	126000	368.22	283.86	1.30	242.55	1.17
8bit 10×10 MM	508000	1487.21	1180.49	1.26	1067.35	1.11
12bit 10×10 MM	1146000	3234.93	2570.84	1.26	2356.41	1.09
16bit 10×10 MM	2040000	5636.27	4606.83	1.22	4185.36	1.10
4bit 20×20 MM	1016000	3153.26	2571.50	1.23	2346.86	1.10
8bit 20×20 MM	4080000	12638.08	10226.60	1.24	9378.26	1.09

TABLE II
SOFTWARE VS BEST FPGA IMPLEMENTATION. ALL UNITS ARE MS

	software	8 AND 8 XOR	speedup
4bit 5×5 MM	659.08	26.42	24.95
8bit 5×5 MM	2684.03	96.61	27.78
4bit 10×10 MM	5391.43	242.55	22.23
8bit 10×10 MM	22031.15	1067.35	20.7
12bit 10×10 MM	49906.86	2356.41	21.18
16bit 10×10 MM	89392.44	4185.36	21.35
4bit 20×20 MM	44466.74	2346.86	18.95
8bit 20×20 MM	179168.64	9378.26	19.10

We will continue to investigate how to handle larger and larger circuits as we accelerate GC in the data center.

The next optimization we plan to target is to improve memory usage by using a separate DDR bank and memory channel for writing back the garble tables. This design will benefit from pipeline of these two parallel data channels. In the current hybrid design, the state machine needs extra states to write back the garbling table, which slows down processing. We also will continue to investigate techniques for BRAM usage and increasing the number of logic gates, as well as the use of URAM in these designs.

Many problems that target FPGAs in the data center will include large data sets that may be accessed in random order. Reorganizing or reordering the data layout with better temporal and spatial locality, and making reuse more efficient, is essential to processing memory bound problems. The lessons learned from these experiments go beyond the particular problem of secure function evaluation and can be applied to other such problems. Getting data to the processing efficiently is a continuing challenge in such systems.

V. CONCLUSIONS AND FUTURE WORK

We have presented the garbling of very large circuits on FPGAs in the data center. Our results, including problem sizes over four million gates, show significant improvement in latency over recently published results. This improvement is due to the use of BRAM to store intermediate results. The techniques developed apply to many problems over large datasets that may not access the data sequentially, and thus must use the BRAM as a user managed cache.

In the future, we plan to investigate how to even more efficiently use the memory hierarchy on an FPGA to support larger problems. We will investigate the optimal organization and management of BRAM as well as the best number of garbled cores to instantiate. We plan to garble even larger problems using multiple AWS nodes enabled FPGAs. We also plan to consider how to efficiently use multiple different types of memory avaiable on FPGAs including UltraRAM, which was introduced in Xilinx Ultrascale+ FPGAs. The memory management solutions will apply to problems with large datasets that are not accessed sequentially and that wish to exploit FPGAs in the data center.

ACKNOWLEDGEMENTS

We would like to thank Xin Fang for his contributions to this research. This research was supported by NSF grant CNS-1717213, and in part by a grant from Amazon Web Services, which supported our experimentation.

REFERENCES

[1] X. Fang, S. Ioannidis, and M. Leeser, "Secure function evaluation using an fpga overlay architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 257–266.

[2] S. U. Hussain and F. Koushanfar, "Fase: Fpga acceleration of secure function evaluation," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 280–288.

[3] Amazon, "Amazon ec2 f1 instances," 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[4] M. Gungor, K. Huang, X. Fang, S. Ioannidis, and M. Leeser, "Garbled circuits in the cloud using fpga enabled nodes," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.

[5] A. Yao, "How to generate and exchange secrets," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986.

[6] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *1st ACM Conference on Electronic Commerce*, 1999.

[7] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *ACM CCS*, 2013.

[8] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2001, pp. 448–457.

[9] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[10] X. Wang and K. Nayak, "FlexSC," 2014. [Online]. Available: https://github.com/wangxiao1254/FlexSC

[11] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM, 1990, pp. 503–513.

[12] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *ASIACRYPT*, 2009.

[13] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, “Tinygarble: Highly compressed and scalable sequential garbled circuits,” in *IEEE S & P*, 2015.

[14] S. U. Hussain, B. D. Rouhani, M. Ghasemzadeh, and F. Koushanfar, “Maxelerator: Fpga accelerator for privacy preserving multiply-accumulate (mac) on cloud servers,” in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 33.

[15] B. D. Rouhani, S. U. Hussain, K. Lauter, and F. Koushanfar, “Redcrypt: Real-time privacy-preserving deep learning inference in clouds using fpgas,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, p. 21, 2018.

[16] X. Fang, “Privacy preserving computations accelerated using fpga overlays,” Ph.D. dissertation, Northeastern University, 2017.

[17] X. Fang, S. Ioannidis, and M. Leeser, “Garbled circuits for preserving privacy in the datacenter,” in *International Workshop on Heterogeneous High-performance Reconfigurable Computing*, 2016.

[18] L. Di Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio, “The role of cad frameworks in heterogeneous fpga-based cloud systems,” in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 423–426.

[19] S. H. Abbas, J.-R. Lee, and Z. Kim, “Fpga-based design and implementation of data acquisition and real-time processing for laser ultrasound propagation,” *International Journal of Aeronautical and Space Sciences*, vol. 17, no. 4, pp. 467–475, 2016.

[20] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, “Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 29–42.

[21] M. Asiatici and P. Ienne, “Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 310–319.