Accelerating Garbled Circuits in the Open Cloud Testbed with Multiple Network-Attached FPGAs

Kai Huang
Google
Sunnyvale, CA
Email: kaihuangjyjs@google.com

Mehmet Gungor, Suranga Handagala, Stratis Ioannidis and Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University, Boston MA
Email: mel@coe.neu.edu

***Abstract*—Field Programmable Gate Arrays are increasingly used in cloud computing to increase the run time performance of applications. For complex applications or applications that operate over large amounts of data, users may want to use more than one FPGA. The challenge is how to map and parallelize applications to a multi-FPGA cloud computing platform such that the problem is partitioned evenly over the FPGAs, memory resources are used effectively, communication is minimized, and speedup is maximized.**

In this research, we build a framework to map Garbled Circuit applications, an implementation of Secure Function Evaluation, to the Open Cloud Testbed, which has FPGA cards attached to computing nodes. The FPGAs are directly connected to 100 GbE switches and can communicate directly through the network; we use the Xilinx UDP stack for this. Preprocessing generates efficient memory allocation and partitioning maps and schedules executions to different FPGAs to minimize communication and maximize processing overlap. This framework achieves close to perfect speedup on a two-FPGA setup compared to a one-FPGA implementation, and can handle large examples that cannot fit on a single FPGA.

1. Introduction

High-performance cloud acceleration architectures should have certain characteristics, including high throughput capacity to process large amounts of data, low processing latency, and the flexibility to keep pace with evolving algorithms and applications. FPGAs are a good choice for the acceleration of cloud computing applications compared with other architectures such as CPUs, GPUs, and ASICs [1].

In this research, we implement Garbled Circuits (GC) across two FPGA nodes in the Open Cloud Testbed, which supports users programming FPGAs that communicate directly through the network. GC is a privacy protocol that is an example of Secure Function Evaluation. It can be applied to large problems to ensure the privacy of user data [2]. GC processes its data in a non-streaming manner which makes partitioning across multiple FPGA designs particularly challenging. We built, designed, and implemented an FPGA overlay to implement Garbled Circuits, which can be downloaded to an FPGA. The design is an extension of our previous work [3], [4], [5] and uses AES cores for encryption. we demonstrate the framework, which includes preprocessing, overlay architecture and partitioning, across several FPGAs on several large problems. The contributions of this research are:

1) Mapping garbled circuits applied to several applications to a cloud platform with network-attached FPGAs, namely the Open Cloud Testbed. The FPGAs communicate directly with each other via a network switch.
2) Careful assignment of data to on-chip and off-chip memory for both local and network data. Memory allocation has a large impact on overall performance for big data applications.
3) Partitioning the workload and minimizing the communication between FPGAs such that both FPGAs process data in parallel. Overlapping communication and computation such that communication is not the bottleneck.
4) Demonstrating near perfect parallelization on two FPGAs in OCT. To do so requires careful memory management and partitioning that is aware of FPGA processing constraints.
5) Mapping large examples that cannot fit on a single FPGA using our framework while improving throughput and achieving acceleration over an optimized software implementation.

The rest of this paper is organized as follows. In Sec. 2, we present background on OCT and GC. Sec. 3 focuses on the tools and techniques used in the project which include partitioning and memory/network allocation. Sec. 4 describes experiments and results, and Sec. 5 concludes with a summary of findings and future work. This paper is based on the first author's PhD dissertation, where more details are available [6].

2. Background

Open Cloud Testbed (OCT). OCT [7], [8] is a research platform that offers FPGA-enhanced nodes to users via the CloudLab framework. It provides the ability for users to develop cloud-based applications that can leverage

FPGAs. CloudLab nodes are bare metal, meaning they are provided without an operating system or any pre-installed software or tools [9].

OCT has AMD/Xilinx Alveo U280 accelerator cards that are directly connected to a 100 GbE network using a 100 GbE data center switch. These network ports are exposed to FPGA users. This enables direct FPGA-to-FPGA communication which results in faster processing times by eliminating the need for processor involvement, thus significantly reducing the latency associated with data transfer between FPGAs. In addition, the U280s are also PCIe-connected to a host processor. This connection is used to transfer initial designs and data from the host to the FPGA, and to retrieve results.

Secure Function Evaluation and Garbled Circuits. Secure Function Evaluation (SFE) guarantees data privacy while preserving data processing functionality. SFE allows one party (the analyst) to evaluate any desirable function over private data from multiple owners. Only the final results are revealed and no relevant party including the analyst obtains the original raw data or information about any intermediate data. SFE can thus enable the data analyst to conduct computations on encrypted data without jeopardizing the privacy of the users. The challenge is that SFE comes with a large amount of additional computational cost. Thus, algorithms executed using SFE usually take much longer than the same computation conducted without security guarantees. Two popular approaches to SFE are Homomorphic Encryption (HE) and Garbled Circuits. In this paper, we focus on GC, which originates from Andrew Yao's paper [10]. Through Yao's protocol, two parties can jointly evaluate a function over private inputs, and learn only the final outcome of this computation. Yao's GC protocol can be extended to multiple-party SFE and attain the above properties for the secure evaluation of any function that can be represented as a Boolean circuit. 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 [13] 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.

Due to their high computational costs, there have been efforts to accelerate both HE [14], [15], [16] and GC with FPGAs. Previous work on accelerating GC has focused on embedded systems [17], or add-ons to processors [18]. Recently researchers have achieved significant speedup by implementing garbled circuits in ASIC hardware [19]. Our research addresses a different point in the design space; namely accelerating big data problems on FPGAs in the cloud.

Garbled Circuit Application Examples. Throughout the paper we apply GC to two examples: Page Rank (PR) and K-means clustering (K-means). Page Rank is a popular algorithm and quickly generates very large GC problems. PR examples are denoted pr_x_y where x is the number of data points and y is the number of iterations. We also use the K-means clustering algorithm [20], [21], [22] as an example where the problem size can be easily scaled. Multiple iterations make the computation longer but do not change the data movement in the system. In the rest of the paper, K-means problems are denoted as kmeans_i_jxk, where i is the number of datapoints, j is the number of classes, and k is the number of iterations. Problem sizes are shown in Table 1.

3. Tools for Multi-FPGA Designs

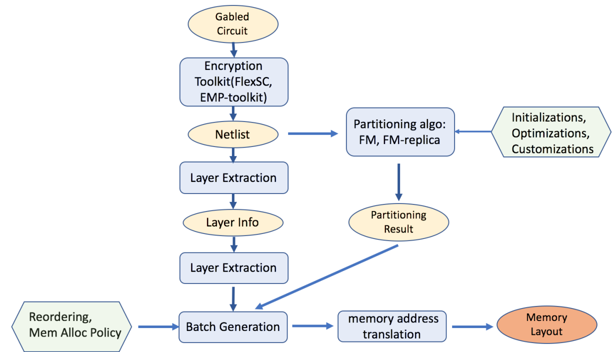


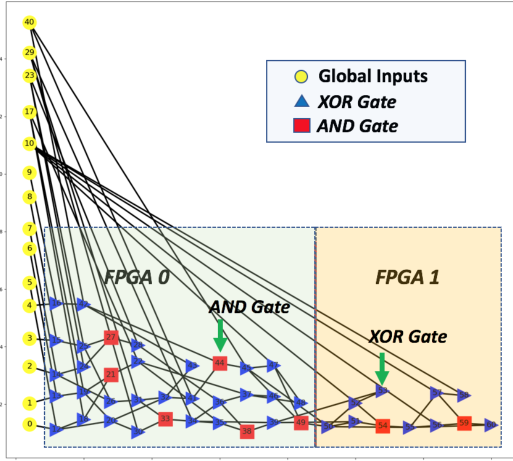
Figure 1: The Garbled Circuits Framework Workflow.

Partitioning and Preprocessing Workflow. The complete workflow is shown in Fig. 1. We use the EMP-tools [23] to generate a netlist, preprocess to determine layer and memory information, partition using the FM algorithm, and then apply a memory allocation policy to generate the final files for computing on multiple FPGAs. Each of these steps is described in more detail below.

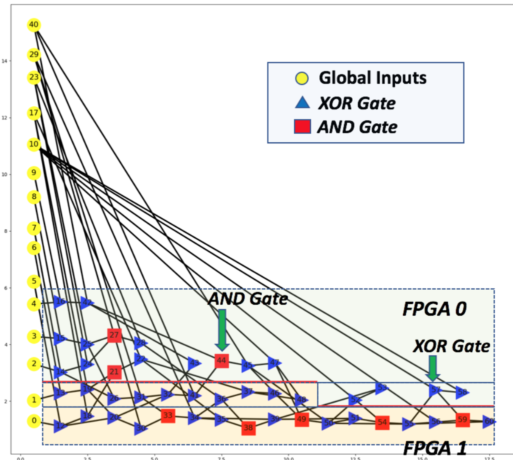
Preprocessing and Partitioning. Preprocessing is needed to extract the workloads, generate the initial memory layout and translate the data addresses. For a multi-FPGA implementation, the workloads for different FPGAs need to be separated. Additionally, an estimate for the memory size needed for data communicated over the network is needed to ensure that there is adequate space in BRAM to store these data. A garbled circuit problem is expressed as a Boolean function that needs to be evaluated. We generate the netlist to evaluate the function from the EMP toolkit [23]. This netlist is processed to extract the Garbled AND gate and free XOR gate operations, which are then organized into layers with no data dependency within each layer. We process the gate list in breadth-first order. Then the execution is grouped into batches where each batch can fit on the FPGA. The number of gates in a layer depends on the problem being garbled, while the number of gates in a batch depends on the hardware implementation. Our implementation realizes 8 garbled AND and 8 XOR gates. Our FPGA architecture implements as many AND and XOR operations in parallel as can be kept busy such that the memory throughput is large enough to saturate these gates. Note that the bottleneck

TABLE 1: K-means Garbled Circuit Problem Size

# data points	# classes	# iterations	Total operations	Total wires	# ops in one iteration	# inputs
100	2	2	2687637	2694037	1343036	6528
100	4	2	5342175	5348575	2671580	6656
100	8	2	10652805	10659205	5325844	6912
1000	2	2	26627497	26691497	13312372	64128
1000	4	2	52950871	53014871	26472584	64256
1000	8	2	105595545	105659545	52750862	64512



(a) Vertical Cut



(b) Horizontal Cut

Figure 2: Horizontal vs. Vertical Cut Partitioning

for the implementation is memory throughput. More gates could be implemented in hardware but they do not improve latency.

Fig. 2 shows the different layers and assignment or operations to logic gates for a small problem. The x-axis is the layer number, and the y-axis is the gate ID. Yellow circles denote input wire IDs, red squares denote AND operations; and blue triangles denote XOR operations. This figure illustrates the dependency of layers in the DAG; within each layer, there are no dependencies by definition. Based on garbled circuit AND and XOR operations, we can map the operation onto the overlay architecture we built on the FPGA for any problem. While generating the netlist, we record statistics to keep track of variable lifetimes and frequency of use to aid in memory allocation. This is done in the same pass as layer and batch extraction.

Memory Allocation Policy. The bottleneck in this and many other designs is getting the data to the processing. Thus, choosing the correct type of memory and memory allocation is important when optimizing system performance. The AMD Alveo U280 includes several types of memory, including BRAM and ultra RAM (URAM) on chip but with different read latencies. In garbled circuits, the netlist is represented by a large amount of monolithic, contiguous memory that gets mapped to either BRAM or URAM modules, and these modules are cascaded. To meet timing requirements, FPGA implementations add extra pipeline stages. BRAM read latency can be 4 or 5 cycles and URAM can reach 8 to 10 cycles. However, even taking this into consideration, the memory throughput of on-chip memory is much larger than that of off-chip memory. It is always recommended to store data that is used frequently in on-chip memory if possible. The problem is finding an efficient way to split data between off-chip memory (HBM or DDR) and on-chip memory (BRAM or URAM) for better memory throughput and system performance. Our goal is to implement a memory allocation policy that has a short processing time, does not require many passes over the data, and improves performance. We studied two different memory allocation algorithms, traversal and threshold. The traversal algorithm traverses the graph in the order that nodes are visited during processing. During that traversal, the type of memory where wires are stored is determined. We explored several different policies during traversal, including (1) a greedy algorithm that stores wires in on-chip memory if space is available, (2) only storing wires used in the next layer, (3) only storing wires used in the next x layers, and (4) store wires used in the next x layers or used over 4 times.

The threshold algorithm combines the lifetime and the frequency of use of wires into a single score, and assigns the wires to on-chip memory according to this score. We set the score of a wire to be `frequency/lifetime`, although other scores can also be applied. We sort these scores and choose a threshold τ. When we traverse the netlist, wires with scores higher than τ are placed in on-chip memory

if space is available, while low-score wires (lower than τ) are placed in off-chip memory. The optimal threshold τ is difficult to determine as it depends on the size of the on-chip memory and the number of wires, as well as the overlaps of lifetimes of wires assigned to on-chip memory. If we set the threshold too high, we assign too few wires to on-chip memory, and some memory space is never used. If the threshold is too small, we may run out of on-chip memory space for more frequently used wires.

We compare results from using the traversal and threshold algorithms. As problem sizes grow and the amount of data stored in off-chip memory increases, the policy used to choose between what goes in on-chip memory and what goes in off-chip memory has increasing importance regarding overall application speed. For large amounts of data, keeping data with many accesses and data with short lifetimes in on-chip memory gives the best results. In some cases, traversal provides the best results, but in more cases, the best result is achieved using thresholding. In addition, thresholding has a shorter computation time. The best thresholding results are from using the top 50% score. Therefore, we adopt the threshold policy for all our experiments.

Network Address Allocation. To support problems across multiple FPGAs, data to be transferred over the network is stored in a separate BRAM module. The required memory size is small compared to that of intermediate wire values and should fit in the available BRAM. For our implementation, we assume that data packets arrive in the order in which they are sent, so preprocessing generates the network data addresses based on the order of data arrivals, and the addresses need not be sent along with the data packet. Preprocessing records the data that needs to be sent to the other FPGA in a map that keeps track of the addresses and distinguishes them from intermediate values received from the same FPGA.

Memory Layout. Data is initially sent to HBM from the host. This includes any global inputs, such as those shown in yellow circles in Figure 2. Global inputs are set to a random 128 bit string. Internal wires and outputs are initialized to zeros and overwritten as processing progresses. Initial netlist memory records the information of the netlist and how the FPGA should process these gates, where to fetch the inputs, and where to write the outputs. Each gate is represented by 16 bytes. The first two 4 byte fields represent the two inputs, the third set of 4 bytes is the output address, and the last 4 bytes identify which type of memory each I/O uses. We use 2 bits as address prefix to denote the address type. '00' represents HBM; '01' represents BRAM; '10' represents URAM; '11' represents BRAM for the network. These are encoded in the last 4 bytes of representation, as is the ID of the Garbled gate that an operation is assigned to. Each batch requires 16x16 bytes in the memory layout. The first 8x16 identifies the Garbled AND gates, the remaining 8x16 bytes identify the XOR gates.

Partitioning. We need to partition the application among the different FPGAs; here we consider two FPGAs. We assume the problem is represented as a graph where the gates, in our case garbled AND and XOR gates, are the nodes and the wires are the edges. We represent the input netlist as a hypergraph where wires are connected to sets of nodes. The goal is to partition this hypergraph into two parts that minimize the cuts between the two, hence minimizing the communications between the FPGAs. Ideally we can hide the communication by overlapping it with computation, but this is not always possible. System performance, which we define as the longest run time on either FPGA, is affected by memory behavior, workload partitioning, communication overhead and problem size. Minimizing communication is one aspect of this optimization problem. It is beneficial to optimize partitioning as a min-cut problem subject to the constraint that the partitions are reasonably balanced.

We consider several algorithms for min-cut, including Kernighan-Lin (KL) [24] and the Fiduccia-Mattheyses (FM) [25], and chose to adopt the FM algorithm, which results in less communications and shorter runtime compared to a baseline algorithm.

Initialization for the FM algorithm plays an important role in the final partitioning result. A general FM algorithm takes a random initialization. In this case, the partitioning algorithm generates a vertical cut, which results in the second FPGA always needing to wait for the results of the first FPGA. In Fig. 2 a, the x axis denotes the layer number and the y axis represents the input/output wires. The direction from left to right is the execution order of the gates. Concerning system performance, this partitioning has no benefit over the one-FPGA design, except to save resources on a single FPGA and to provide a larger memory footprint. Instead, we need the horizontal cut from figure b, so that gates from each layer are assigned to different FPGAs. We customized the initialization for the FM algorithm such that each layer is equally assigned to create two initial partitions. We observe that among each layer, there are strongly connected components about the gates' inputs and outputs. Thus, When we initialize, cross-layer dependency is also taken into account. If the two inputs of a gate come from the same partition, this gate will also be assigned to that partition. If the two inputs of the gate come from different partitions, this gate will be assigned to either partition based on a seed. That leaves room for the FM algorithm to improve the final partitioning results. The first layer is equally partitioned into two parts.

4. Experiments and Results

We provisioned the AMD Alveo U280 FPGAs in OCT through CloudLab. For OS we used Ubuntu 18.04, and the Xilinx XRT version 2021.2 was used. This system setup was used for all of the experiments.

4.1. One FPGA Experimental Results versus Software Implementation

An important consideration in performance is how memory is allocated. In all our designs, the processors transfer

TABLE 2: Application latency for Different BRAM Sizes with the Same Design

K-Means	HBM + 50k BRAM (ms)	HBM + 100k BRAM (ms)	HBM + 200k BRAM (ms)
100_2x2	103	125	109
100_4x2	198	244	218
100_8x2	389	575	462
1000_2x2	1382	1603	1468
1000_4x2	3199	3338	3138

TABLE 3: Clock speeds as BRAM size increases

HBM + 50k BRAM (ms)	HBM + 100k BRAM (ms)	HBM + 200k BRAM (ms)
266MHz	214 MHz	167 MHz

the initial information regarding the netlist and input values to HBM. Intermediate values may be stored in one of several different types of memory Including BRAM, URAM, and HBM. Table 2 shows the overall latency in milliseconds for different Garbled Circuits experiments when the size of the problem and the size of the BRAM is varied. Here the application being garbled is K-means which was chosen because it can easily be scaled in size. For the smaller sizes of 100_2x2 and 100_4x2, the data fits completely into BRAM. The largest problem we report on, 1000_4x2, has 1000 data points, with four classes and runs for two iterations. It generates more than 26 million intermediate keys. Clock speeds for these designs are shown in Table 3. The drop in performance for larger BRAMs is a result of the clock speed drop and no other effect. When a combination of BRAM and HBM are required, the performance is more complicated. As expected, larger amounts of BRAM result in better overall performance, even with a clock speed drop, due to the shorter latency to access data present in the BRAM. The clock speed drop for larger amounts of BRAM in conjunction with HBM is due to routing congestion. When 50K of BRAM is used, the FPGA design and BRAM are placed close to the HBM. The design with 50K of BRAM uses less than 10% of the available BRAM, while the design with 400k uses 70%. The FPGA design remains the same and consumes fewer than 10% of other resources. The distribution of BRAM across the chip results in routing congestion and a slower clock rate.

Table 4 compares performance among Garbled Circuits Designs of K-means that use HBM alone, HBM plus BRAM and HBM plus BRAM plus URAM. All results are given in milliseconds of total latency. When BRAM and URAM are included, 100 KBytes of URAM and 100 KBytes of BRAM were used. Our experiments show that HBM plus BRAM delivers the best results. Adding URAM did not improve performance. While URAM is an excellent resource, it is not effective to use in conjunction with both HBM and BRAM for our garbled circuit hardware design. That is mainly due to the fact that a large size of URAM leads to a significant clock frequency drop.

For the one FPGA implementation, judicial choice of memory has a big effect on performance. Using over 75% or higher BRAM or using BRAM together with a big chunk of URAM can cause a frequency drop to less than 100MHz. Larger BRAM benefits the system run time simply by providing more memory throughput as shown in Table 2. However, even with improving the memory layout, the software implementation provided by the EMP-toolkit (written in C++), performs better for large examples. There are several reasons for this. The most important is that AES encryption and decryption are supported in hardware for microprocessors from Intel [26] and Advanced Micro Devices. They provide AVX and AESNI instruction extensions. The CPU generally runs at over 2.5 Ghz, compared to the FPGA frequency running at 200 Mhz. It is impressive that we are able to achieve speeds close to the software speed given this divergence. However, it is difficult for the FPGA to exceed the CPU processing speed unless we can improve the memory bandwidth for off-chip memory to keep more gates busy. Without improved bandwidth, the parallelism available on the FPGA cannot compensate for the disadvantage of the clock frequency. When the problem size gets large, the number of wires can reach tens of millions. Even with reuse, the number of wires stored in BRAM is much smaller than the total number of wires. For a typical garbled circuit problem, the first several layers are extremely large compared to later layers, and the BRAM can store only a portion of wires even if those wires will be used in the next layer. A large number of wires still get assigned to off-chip memory. The system performance will thus be affected by off-chip memory behavior. To improve performance, we looked at more than one FPGA which both increases parallel processing and allows us to access more memory in parallel.

4.2. Mapping to Two FPGAs

For the experiments that make use of two FPGAs, the page rank and k-means applications that were tested are listed in Table 5. The applications are different implementations secured by Garbled Circuits and the size of the problem is varied. We record the total number of gates and the communication requests from each FPGA. Since each input wire is 128 bits, the request here denotes how many pieces of 128 bit data need to be transferred between FPGAs.

We run the applications on two nodes of OCT, each node equipped with one FPGA. We run host python code to move data to FPGA off-chip memory, set up the IP address port number, and populate the hardware socket tables. We pass the top-level kernel parameters to the FPGA including the hardware mode, destination_id, time_between_packets_to_send, etc. Finally, we pass the

TABLE 4: Latency of Designs with different types of memory

K-Means	EMP-toolkit(ms)	HBM,BRAM,URAM(ms)	HBM(ms)	HBM,BRAM(ms)
100_2x2	189	178	181	109
100_4x2	351	356	361	218
100_8x2	651	706	724	462
1000_2x2	1512	2053	1806	1468
1000_4x2	2900	4184	3543	3138

TABLE 5: Amount of Communication Between FPGAs

Application	**Total operations**	**Requests from FPGA1**	**Requests from FPGA0**
pr_10_1	65528	0	0
pr_10_5	327640	0	0
pr_100_1	1209825	0	0
kmeans_20_4	555,036	34	37
kmeans_50_4	1,347,370	34	37
kmeans_100_4	2,671,580	34	37
kmeans_100_8	5,235,844	128	126
kmeans_1000_2	13,312,374	1	16
kmeans_1000_4	26,472,584	34	37

start signals to two FPGAs with the Dask [27] scheduler and workers. The hardware will launch the hardware handshaking first then start to process the netlist. All data transfer during runtime between FPGAs makes use of the direct connection. When the entire netlist has been processed, the total number of clock cycles is recorded for each FPGA and returned to the host. We rely on python support from the Xilinx network examples to set up the UDP network stack parameters, and we directly record the hardware timing. The hardware timingrecords the time from the moment that the two FPGAs start to execute the netlist instructions until the end of the computation.

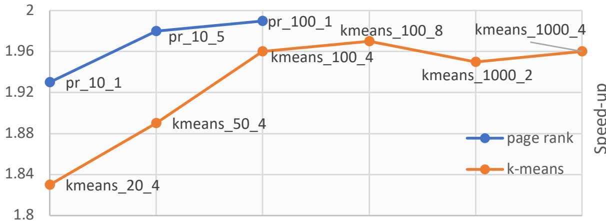


Figure 3: Two-FPGA system speed up over one FPGA implementation

Table 6 shows the run time for two FPGAs, and Fig. 3 shows the speedup of two FPGAs over one FPGA running the same design. This was calculated by dividing the runtime of a single FPGA by the slowest runtime for the two FPGAs. Table 7 compares the two FPGA results to the software implementation, with FPGA frequency at 200 Mhz. With the two FPGA system, we achieve at least 1.8x speedup over the single FPGA system, and acceleration of around 1.4 times over software. To achieve these results, we ensured that partitions were balanced by constraining the size of each partition to be larger than 40% and smaller than 60% of the total operations. As the problem size gets larger, the partitioner nearly equally assigns the operations to each of the two FPGAs. The page rank examples can be perfectly assigned to two FPGAs without any network overhead, thus the speedup is approximately 2. If we examine the kmeans examples with network costs, the speedup is around 1.9x to 2x compared to the one FPGA implementation.

For larger examples, the partitioning algorithm optimizes and minimizes the network cost. Also, the communication cost is overlapped with computations, which enables us to achieve near perfect speedup with a two-FPGA system in a cloud computing environment.

We also explored one extremely large example that does not fit on a single FPGA because the netlist information does not fit in off-chip memory if we allow space for intermediate data as well. The largest example netlist is roughly 3.5 GB, so we have to split it across two FPGAs. After partitioning, the netlist in FPGA0 takes a total of 7 banks of memory and the netlist in FPGA1 takes 6 banks, where the size of each bank is 256 MB. We are able to run the example successfully on this two-FPGA system with approximately 1.7x speedup over the software implementation. Multiple FPGA implementations are promising approaches to garbling large problems; we intend to continue to expand on this area of research by considering more than two FPGAs.

5. Conclusions and Future Work

We have presented a heterogeneous reconfigurable computing framework using an FPGA overlay architecture for a non-streaming application, specifically the garbled circuits problem. The complete workflow presented here ensures the user can implement and accelerate their application without knowledge of either hardware development or the secure function evaluation protocol. Once the base processing elements are available in the overlay, new problems can be mapped to the same overlay without reprogramming. The

TABLE 6: Comparison of runtime (in clock cycles) between one-FPGA and two-FPGA runs, with achieved speedup.

Application	One FPGA run time	FPGA0 run time	FPGA1 run time	Speed-up
pr_10_1	1,138,453	588,303	589,453	1.93
pr_10_5	5,589,602	2,823,857	2,823,026	1.98
pr_100_1	21,335,125	10,688,206	10,673,430	1.99
kmeans_20_4	9,513,658	5,204,968	4,665,416	1.83
kmeans_50_4	23,360,348	12,354,741	11,664,804	1.89
kmeans_100_4	47,604,557	24,284,454	23,295,225	1.96
kmeans_100_8	94,796,337	48,100,878	46,687,401	1.97
kmeans_1000_2	238,735,325	122,240,085	116,439,651	1.95
kmeans_1000_4	469,211,537	238,795,607	232,218,775	1.96
kmeans_10000_2	N/A	1,204,340,391	1,150,350,240	N/A

TABLE 7: Run time of two-FPGA Design vs Software(s)

Application	Software run time	2-FPGA run time	Speed-up
pr_10_1	0.016	0.0023	6.90
pr_10_5	0.070	0.011	6.10
kmeans_20_4	0.039	0.021	1.86
kmeans_50_4	0.091	0.050	1.83
kmeans_100_4	0.18	0.097	1.86
kmeans_100_8	0.37	0.193	1.92
kmeans_1000_2	0.80	0.49	1.63
kmeans_1000_4	1.75	0.95	1.84
kmeans_10000_2	8.13	4.82	1.69

preprocessing workflow includes a problem parser, layer and batch generation, memory allocation policy, and initial memory layout generation tools. The framework is configurable for different garbled circuit problems and only the initial memory content, consisting of wire addresses and hardware gate mapping, needs to be regenerated for a different problem. These tools help explore the parallelism for any Garbled Circuit problem. Assuming an overlay-based architecture, the same tools and approaches presented here can be applied to other problems. This includes layer and batch extraction as well as partitioning. We applied FM with customized initialization to our problem to optimize partitioning to more than one FPGA.

Our framework is implemented in OCT, which allows users to program FPGAs that are directly connected to the network. We integrate our design with the Xilinx UDP network stack to support direct FPGA-to-FPGA communication through the network while bypassing the processor. We are able to achieve near-perfect acceleration on two FPGAs for the GC example. In addition, we are able to run examples that are too large to be mapped to a single FPGA.

In the future, we plan to investigate mapping garbled circuits to more than two FPGAs. This will involve partitioning into an arbitrary number of partitions. We also plan to continue to investigate the different types of memories available and how to best make use of them. In this and many other big data applications, the challenge is getting the data to the processing. We have space for many more garbling gates in our overlay, but we are not able to keep them supplied with data, so they would remain idle if they were instantiated. We specifically plan to investigate how to improve the memory read bandwidth to increase parallelism.

Network efficiency could also be improved. We use 128 bits for each piece of intermediate data, and the network state machine sends 128 bits as soon as the bits are generated. However, it is preferable to concatenate data from several memory addresses. This saves network overhead by removing unnecessary padding and headers, so as to increase the network throughput. The disadvantage is that the concatenation makes the hardware design more complicated. Additionally, if the problem size is not large, concatenation may make the system run slower. Nevertheless, the efficiency of packing multiple intermediate values should be important for scenarios where the communication workload is heavy.

Acknowledgments

This research was funded in part by National Science Foundation (NSF) Grant SATC 1717213. The Open Cloud Testbed is funded by NSF grants CNS-1925464, CNS-1925504, and CNS-1925658. All opinions and statements in this publication are of the authors and do not represent NSF positions.

References

[1] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M. Herbordt, H. Shahzad, P. Hofste, B. Ringlein, J. Szefer, A. Sanaullah, and R. Tessier, "The future of fpga acceleration in datacenters and the cloud," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 3, feb 2022. [Online]. Available: https://doi.org/10.1145/3506713

[2] X. Wang, S. Ranellucci, and J. Katz, "Global-scale secure multiparty computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 39–56. [Online]. Available: https://doi.org/10.1145/3133956.3133979

[3] K. Huang, M. Gungor, 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.

[4] X. Fang, S. Ioannidis, and M. Leeser, "SIFO: secure computational infrastructure using FPGA overlays," *International Journal of Reconfigurable Computing*, vol. 2019, p. 1439763, Dec 2019. [Online]. Available: https://doi.org/10.1155/2019/1439763

[5] ——, "Secure function evaluation using an FPGA overlay architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 257–266. [Online]. Available: https://doi.org/10.1145/3020078.3021746

[6] K. Huang, "Partitioning data across multiple, network connected FPGAs with high bandwidth memory to accelerate non-streaming applications," Ph.D. dissertation, Northeastern University, Boston, MA, USA, 2022.

[7] M. Leeser, S. Handagala, and M. Zink, "FPGAs in the Cloud," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 72–76, 2021.

[8] S. Handagala, M. Leeser, K. Patle, and M. Zink, "Network Attached FPGAs in the Open Cloud Testbed (OCT)," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2022, pp. 1–6.

[9] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[10] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 1986, pp. 162–167.

[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*, ser. STOC '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 503–513. [Online]. Available: https://doi.org/10.1145/100216.100287

[12] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *Advances in Cryptology – ASIACRYPT 2009*, M. Matsui, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 250–267.

[13] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*. Springer, 2008, pp. 486–498.

[14] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: an accelerator for homomorphic encryption on the amazon AWS FPGA," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.

[15] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-accelerated homomorphic encryption co-processor," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 193–206, 2016.

[16] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 882–895.

[17] 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)*, 2019, pp. 280–288.

[18] E. M. Songhori, M. S. Riazi, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "Arm2gc: Succinct garbled processor for secure computation," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[19] J. Mo, J. Gopinath, and B. Reagen, "HAAC: a hardware-software co-design to accelerate garbled circuits," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[20] J. Bhimani, M. Leeser, and N. Mi, "Accelerating K-Means clustering with parallel implementations and GPU computing," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015, pp. 1–6.

[21] M. Beye, Z. Erkin, and R. L. Lagendijk, "Efficient privacy preserving k-means clustering in a three-party setting," in *2011 IEEE International Workshop on Information Forensics and Security*. IEEE, 2011, pp. 1–6.

[22] M. Leeser, J. Theiler, M. Estlick, and J. J. Szymanski, "Design tradeoffs in a hardware implementation of the k-means clustering algorithm," in *Proceedings of the 2000 IEEE Sensor Array and Multichannel Signal Processing Workshop. SAM 2000 (Cat. No. 00EX410)*. IEEE, 2000, pp. 520–524.

[23] J. K. Xiao Wang, Alex J. Malozemoff. (2016) EMP-toolkit: Efficient MultiParty computation toolkit. [Online]. Available: https://github.com/emp-toolkit

[24] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[25] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th Design Automation Conference*, ser. DAC '82. IEEE Press, Jan. 1982, pp. 175–181.

[26] Intel. (2016) Intel 64 and ia-32 architectures software developer's manual. Intel. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671200

[27] (2022) Dask. Anaconda, Inc. [Online]. Available: https://docs.dask.org/en/stable/