Optimizing Use of Different Types of Memory for FPGAs in High Performance Computing

Kai Huang
Dept. of ECE
Northeastern University
Boston, USA
huang.kai1@northeastern.edu

Mehmet Gungor
Dept. of ECE
Northeastern University
Boston, USA
gungor.m@northeastern.edu

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

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

***Abstract*—Accelerators such as Field Programmable Gate Arrays (FPGAs) are increasingly used in high performance computing, and the problems they are applied to process larger and larger amounts of data. FPGA manufacturers have added new types of memory on chip to help ease the memory bottleneck; however, the burden is on the designer to determine how data is allocated to different memory types. We study the use of ultraRAM for a graph application running on Amazon Web Services (AWS) that generates a large amount of intermediate data that is not subsequently accessed sequentially. We investigate different algorithms for mapping data to ultraRAM. Our results show that use of ultraRAM can speed up overall application run time by a factor of 3 or more. Maximizing the amount of ultraRAM used produces the best results, and as problem size grows, judiciously assigning data to ultraRAM vs. DDR results in better performance.**

***Index Terms*—FPGA, High Performance Computing, Big Data, AWS**

I. Introduction

There is growing interest in deploying FPGAs to accelerate applications in High Performance Computing (HPC) and cloud environments [1]–[4]. Many of the target applications fetch or generate large amounts of data and increasingly, the bottleneck to accelerating these designs is the data fetch time. This is particularly true of applications where the data is not accessed in sequential order, and thus streaming interfaces cannot be exploited. To address this problem, FPGA vendors are increasing the amount and types of memory integrated into high end FPGA cards that target high performance applications. Block RAM (BRAM) embedded with the FPGA fabric has been available for many years. Off-chip DRAM has much larger capacity than BRAM, but is orders of magnitude slower to access. In the past few years, Xilinx has added ultraRAM to their FPGAs [5], which is larger than BRAM but has similar access times, and more recently High Bandwidth Memory (HBM). This creates an additional burden for the application developer, who now has to determine where best to store data in order to minimize the amount of time spent fetching data and maximize acceleration. We investigate an application that generates a large amount of data, and use the FPGAs provided by Amazon [6] to accelerate it. We investigate tradeoffs between using ultraRAM and DDR on AWS F1 instances, including different algorithms for allocating data to memory, and show that intelligent assignment of data to different types of memory can provide distinct advantages. Traditional caching algorithms used in processors are not the best choice for memory allocation on FPGAs, and matching the algorithm for memory allocation to the application can result in significant time savings.

The main contributions of this paper is a thorough exploration of the use of ultraRAM vs. DDR for a specific application, garbling K-means clustering. The study allows us to explore different algorithms for allocating data to ultraRAM, along with how behavior changes as the amount of data generated by the application grows. Tuning algorithm to application and size will result in the best performance. The rest of the paper is organized as follows. We present background in Sec. II, our methodology in Sec. III, results in Sec. IV, and conclude in Sec. V.

This research is supported in part by the National Science Foundation under grant CNS-1717213

II. Background

A. FPGA Memory Types

We target FPGAs in the Amazon cloud, which are Xilinx FPGAs that contain several different types of memory. The FPGA is a Xilinx XCVU9P which contains 33.8 MBytes of ultraRAM and 9.5 MBytes of Block RAM. In addition there is 64 GBytes of external Dual Data Rate (DDR) memory. As is typical in memory hierarchies, the Block RAM is the smallest and fastest to access, ultraRAM has similar access time to BRAM, and DDR is the largest but slowest. We do not consider BRAM further. In this section we discuss DDR and ultraRAM in more detail.

a) DDR: Double Data Rate (DDR) Synchronous Dynamic Random-Access Memory (SDRAM) is commonly used in computer systems. AWS uses DDR4 SDRAM; each FPGA card has 64 GigaBytes of external DDR. On AWS, we have measured 52 clock cycles for DDR read and 32 clock cycles for DDR write, making it more than an order of magnitude slower than accessing ultraRAM.

b) ultraRAM: FPGAs have included on-chip memory in the form of block RAM and distributed RAM for years, but these types of on-chip memory can support at most tens of megabytes. Traditionally, when more data is required external

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

memory was used, but this increases the latency for memory fetching. In 2016, Xilinx introduced ultraRAM into their high-end Virtex UltraScale+ FPGAs. Every ultraRAM block is a dual-port synchronous 288Kb RAM with fixed configuration of 4,096 deep and 72 bits wide and two ports, which exhibit some unexpected behavior [7]. Port A and B share the same clock. Each port can independently perform either one read or one write operation. Within a single cycle of the external clock, the Port A operation always completes before Port B. When both ports perform a write operation in the same clock cycle with the same address (address collision), the Port B write takes effect because the write on Port A is overwritten. When Port A performs a read while Port B performs a write with the same address, Port A gets the old data and then the new data at Port B is written. For our design, we use one port as a read port and one port as a write port. We do not read and write to the same address at the same time.

Data is not automatically assigned to ultraRAM; this is under the control of the designer. There are several ways to initiate a Xilinx ultraRAM block instance in hardware: 1) developers can instantiate a RAM inference template with a parameter setting the memory type to ultraRAM, 2) developers can instantiate a device primitive if developers want to have tighter control over how individual components are connected, or 3) developers can use the XPM template and let the Vivado tools synthesize the source and create the appropriate memory arrays. We use the third approach. Note that, after setting the memory type, size and behavior ordering of the ports, developers need to be careful about the latency of these ports to meet synthesis constraints. Once these are met, we observed a 3 clock cycle latency for reading and writing ultraRAM. Applications can benefit from the use of ultraRAM due to its large size and low latency.

B. AWS infrastructure

We use Amazon Web Services (AWS), which provides many resources for cloud computing, including f1 instances that include FPGAs from Xilinx to enable delivery of custom hardware acceleration [6]. We use the f1.2xlarge with Virtex Ultrascale+ XCVU9p FPGAs. This part includes 33.8 MBytes of ultraRAM. The FPGA board includes 4x16 GB external DDR4 memory.

Amazon provides complete hardware development and software development toolkits targeting the FPGA including the development environment, simulation, build and AFI creation scripts. The development environment contains the register-transfer level (RTL) hardware infrastructure that is built with Xilinx IPs, and source code based on Advanced eXtensible Interface (AXI) protocol. AWS-FPGA hardware infrastructure connects the FPGA board, including external DDR memory, to the host processor through a PCIe Gen3 bus. The Xillinx interconnect IP with AXI protocol in the hardware design enables data movement between host memory, FPGA on-chip memory (including ultraRAM) and DDR memory on the FPGA board. 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

Several researchers investigate how best to access memory on FPGAs for efficient processing. In DynaBurst [8], on-chip memory is used to overcome DDR row conflicts by organizing the output data in on-chip memory before writing it to DDR sequentially. This reduces DDR row conflicts and adapts the burst length for each request to fully use the bandwidth. DynaBurst does runtime memory management, while our approach is done statically before processing. This paper only considers BRAM and DDR; we also consider ultraRAM as on-chip memory. UltraRAM and BRAM is used in research [9] that proposes a two-level vertex caching method to improve the performance of processing graphs on FPGAs by reducing the data communication between FPGA and DDR. UltraRAM is used as a Level 2 cache and BRAM is used as a level 1 cache for vertex caching. This method falls short for large sparse graphs. It also uses traditional caching algorithms, while we try to exploit the FPGA architecture to make best use of the available FPGA memory. More recently, researchers have investigated the use of High Bandwidth Memory (HBM) on FPGAS to address the speed limitations of the communication between FPGA and DRAM [10]. They showed that HBM can increase overall system performance. The research also shows how to reach near peak performance on HBM systems. In future work, we plan to consider HBM to improve system performance. HBM is currently not available on AWS f1 instances.

III. Methodology

A. Problem Abstraction

In this study, we investigate memory usage in a graph based algorithm, accelerated on AWS F1 instances. We specifically investigate K-means clustering implemented with garbled circuits [11], as this allows us to show how data size and memory allocation times scale as problems grow. In this application, the problem to be evaluated is represented as a Directed Acyclic Graph (DAG), where each node represents either a garbled AND or garbled XOR gate, and has two inputs and one output wire. These wires represent encrypted values and are 128 bits wide. Values for input wires are transferred to DDR memory attached to the FPGA from the host. Intermediate values are generated as the application runs. We use 32 bit values to represent addresses for wire values. One of these bits is reserved to encode the type of memory used, in this case ultraRAM or DDR, leaving 31 bits for addressing wires, which allows us to grow our design up to 2 billion wires.

Note that, while some details are specific to the application we are running, many other big data problems have similar access patterns with a set of inputs and a set of intermediate values generated as the application runs that can grow quite large. Also, note that we process the graph in breadth first order. Intermediate values are generated out of order in a K-means stage, and may be used in the next layer or not for

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

many layers. Other applications, such as sparse matrix and graph processing, exhibit similar memory access patterns.

B. Allocating wires between ultraRAM and DDR

The hardware controller needs to store intermediate data in either on-chip memory (ultraRAM) or external storage (DDR). For small examples, the design can store all the intermediate data in ultraRAM. For applications that can be perfectly pipelined, only registers between pipeline stages are required to complete the critical data path. However, for applications with non-sequential access to data memory needs to be carefully allocated. For large examples that contain millions of operations and data points, it is not possible to assign all intermediate values to ultraRAM. In this research we investigate different methods for assigning intermediate data to this memory hierarchy which can significantly impact the overall performance of hardware acceleration. Note that we only consider ultraRAM and DDR here. Also using BRAM for intermediate values is the subject of future work. One issue is how to determine at runtime where to find a particular value. Currently we use a single bit of address; using two bits makes the address space smaller that what we wish to consider.

In the AWS hardware infrastructure, AXI interconnect is used to connect the external memory (DDR) with custom user logic. The user logic requests the data through the AXI bus and the latency is usually around 52 cycles. In contrast, ultraRAM is accessed through primitives and the access time is about 3 clock cycles depending on the template settings.

C. Memory Allocation Algorithms

Our application can be abstracted as a netlist containing gates each with input and output wires, constructing a DAG. The DAG consists of several layers, that can be identified recursively via a breadth-first traversal of the gates/nodes of the DAG. The information carried on each wire is 128 bits wide in our application. From the topology of the graph we can determine the number of times a wire is used (frequency). We define a wire's lifetime to be the difference between the layer number of the last time the wire is used and the layer number where it is generated. We also consider whether a wire is used soon after it is generated, independent of its lifetime, in our traversal algorithms. Note that most layers cannot be implemented at the same time due to hardware constraints, and thus a layer is implemented as a sequence of batches, where a batch is defined by the the number of gates physically realized in hardware. Especially early in processing, there may be hundreds of batches to implement a specific layer. We do not consider the position of a gate in a layer in terms of batch number, although such a consideration could result in better memory allocation for large examples. We have developed two main classes of memory allocation algorithm based on frequency, lifetime and layer information. These two algorithm types of algorithm, *traversal* and *threshold*, are described below. First we describe preprocessing used by all the algorithms.

1) Preprocessing: The algorithms visit each gate in the order of gate execution, which in our application is breadth-first (starting from gates accessing inputs). Before allocating each wire to a memory location, we preprocess the netlist as follows. First, the layer number for each gate is generated during a breadth-first traversal. All gates assigned to layer n can be computed in parallel given sufficient hardware resources. The inputs required of these gates are generated in layers 0 to $n-1$. Second, preprocessing keeps track of the following information regarding each wire: (a) how many times the output wire will be used (i.e., how many gates use it as input, (b) in which of the succeding layers it appears (maintained only for traversal algorithms), and (c) what is its lifetime (maintained only for threshold algorithms). This information is computed while generating the layer numbers; thus only a single breadth-first pass over the netlist is required for preprocessing.

TABLE I: frequency of wires for K-means 100 points, 8 classes

frequency	num of wires	percentage
1	2646610	0.498
2	1691585	0.319
3	868713	0.164
4	3616	0.001
5	4081	0.001
⋮	⋮	⋮
100	8	0
101	8	0
200	16	0
400	496	0

Table I, lists the number of times wires are used for one example we study. Approximately 80% of the wires will only be used once or twice. It is important to reuse these memory locations in future layers. We maintain a queue to record the available ultraRAM locations and we also check, while processing the netlist, when we can free a specific location if a wire will never be used again. If there are no available ultraRAM locations, we will store wires in DDR, the external storage. Our goal is to store as many wires as possible in ultraRAM.

2) Traversal: 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 a greedy algorithm that stores wires in ultraRAM if space is available, storing wires with short life-times, and storing the most frequently used wires. Specifically, we considered four policies:

- policy 1: store wires in ultraRAM if ultraRAM is not full
- policy 2: store wires used in the next layer only
- policy 3: store wires used in the next x layers, where $x = 5$.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

Algorithm 1 Traversal Algorithm

```
generate the layer number each wire is created and last used, record its frequency and whether it is used in the next x layers
maintain an empty queue
record the max used ultraRAM location p
for operation s in netlist do
    for input i in inputs do
        if i will not be used again then
            free its location if it is in ultraRAM, add the address of i to queue
        end if
    end for
    if output s will be used in next x layers or will be used over y times then
        if queue is not empty then
            pop an address for output s
        else if max used ultraRAM location smaller than ultraRAM size and queue is empty then
            use address p for s and increase max used ultraRAM location p
        else
            store s to DDR
        end if
    end if
end for
```

- policy 4: store wires used in the next x layers or used over 4 times.

The pseudo-code for the traversal algorithm is shown in Algorithm 1. The different policies are implemented by changing the values of x and y. We investigated different choices and recorded statistics including how many wires will be stored in ultraRAM and how many total clock cycles will be spent accessing memory.

3) Threshold: Our alternative class of algorithms combines the lifetime and the frequency of the wires into a single score associated with each wire. We assign the wires to ultraRAM according to this score. The pseudo-code is shown in Algorithm 2. We experimented with setting the score of a wire to be to `frequency/lifetime`, although other scores can also be applied. We sort these scores and choose a threshold τ. When we traverse the netlist, we assign each wire based on whether or not its score is over the threshold: wires with scores higher than τ are placed in ultraRAM, if a position is available, while low-score wires (lower than τ) are placed by default in the DDR, even if a position is available for them in ultraRAM. We determine the theshold τ based on either size of available ultraRAM or percent of wires to fit in ultraRAM, as described below. Note that we can bin rather than sort the scores for a more efficient algorithm.

The optimal threshold τ is difficult to calculate efficiently. It depends on the size of the ultraRAM and the number of wires, as well as the overlaps of lifetimes of wires assigned to ultraRAM. If we set the threshold too high, we assign too few

Algorithm 2 Threshold Algorithm

```
generate the layer number each wire is created and last used, record its frequency and lifetime
calculate the score using frequency/lifetime
Sort all wires according to their scores in decreasing order
determine a threshold τ
maintain an empty queue
record the max used ultraRAM location p
for operation s in netlist do
    for input i in inputs do
        if i will not be used in the future then
            free its location if it is in ultraRAM, add the address of i to queue
        end if
    end for
    if output s ≥ τ and there is space in ultraRAM then
        store s to ultraRAM
    else
        store s to DDR
    end if
end for
```

wires to ultraRAM and some memory spaces are never used. If the threshold is too small, we may assign wires that will not be used for several layers and we run out of ultraRAM space. We experimented with setting the threshold τ as follows: for $f \geq 1$, and S the ultraRAM size, we select τ so that the number of wires with score $\geq \tau$ is $f \cdot S$. In our experiments, we select f to be 1.5, 2, 4, and 8. As a second heuristic, for $f' \in [0, 1]$, and W the total number of wires, we select the threshold τ so that the number of wires with score $\geq \tau$ is $f' \cdot W$. In our experiments, we select f' to be $1/4$, $1/2$, and $3/4$. We record the statistics for these different thresholding implementations and compare them to the traversal algorithm.

IV. Experiments and Results

We use the XCVU9P FPGA which includes 270 Mbits of ultraRAM. Given our 128 bit data width, we are able to create an ultraRAM memory array with 2 million distinct addresses. We garbled different versions of the K-means clustering algorithm. Different problem sizes we investigated are shown in Table II.

We can calculate the theoretical lower and upper bounds of the number of clock cycles required for memory accesses in our application. The lower bound assumes that all data points are stored in ultraRAM and the upper bound assumes that they are all stored in DDR. These bounds for different problem sizes are shown in Table III. Note, that for smaller problems all data can fit in ultraRAM. These cases do not achieve the lower bound for a couple of reasons. The main one is that input wires are stored in DDR from the host, and reading that first layer of wires adds memory access clock cycles.

In our results we report total number of clock cycles used to access memory. As memory accesses and processing are overlapped it is not immediately obvious how this translates

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

TABLE II: K-means Problem Size

# data points	# classes	# iterations	total operations	total wires	# of 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

TABLE III: Lower and Upper bounds

data points	classes	lower bound	upper bound
100	2	8.06E+06	1.42E+08
100	4	1.60E+07	2.83E+08
100	8	3.20E+07	5.65E+08
1000	2	7.99E+07	1.41E+09
1000	4	1.59E+08	2.81E+09
1000	8	3.17E+08	5.59E+09

to overall performance savings. We investigated the savings we expect to see comparing storing all values in ultraRAM compared to storing all values in DDR. Even with overlap of processing and memory fetches, our experiments show a 4.05 times improvement in overall execution time assuming all data can be stored in ultraRAM. This is an upper bound on the improvement we expect to see in application processing. We believe that over 3x speedup is realistic.

For our implementation, we use K-means algorithm as a practical application example, where the problem size can scale easily. The operations in one iteration will be repeated. Multiple iterations make the computation longer but do not change the data movement in the system. Problem sizes are shown in Table II.

For the traversal algorithm, we set the total ultraRAM size to 2,000,000 wires. Total number of wires stored and total number of memory cycles needed are shown in Tables IV and V, respectively. Results for the threshold algorithms are presented in Table VI. The best result for each set of experiments is highlighted in red. For thresholding, we see no difference when we set f' to 1/4, 1/2 or 3/4, so we report results for 1/4.

In Table VII, we compare the best traversal algorithm results with the best threshold algorithm results. For smaller examples, with all wires fitting into ultraRAM, there is no difference between algorithms and policies. For larger examples the traversal algorithm works best. In genera, the more wires that are stored in ultraRAM, the fewer memory access cycles required. In the largest example we ran, the policy that supports traversal with wires used in the next several layers and wires used frequently produces the most efficient allocation. This is expected since, when the amount of data generated is much larger than the ultraRAM size, if ultraRAM is full, any wire generated will be sent to DDR. Given that majority of the wires in our application are used less than 3 times, the shorter the average time wires reside in ultraRAM, the more wires we can assign and thus the less total memory access time is needed. We plan to continue to investigate larger examples to confirm this trend.

A. Discussion

As different memory types get added to FPGA accelerators, efficient use of such memories becomes increasingly important, especially in big data applications. In this research, we consider ultraRAM and DDR. AWS F1 instances have BRAM as well. A future direction is to consider putting the most frequently accessed wires in BRAM, the wires with a short lifetime in ultraRAM, and the remaining wires in DDR. One issue is how to identify where to find a wire. We currently use the first bit of memory address to indicate whether a wire is stored in ultraRAM or DDR, which keeps the hardware controller interface straightforward. Adding another bit to indicate memory location would halve the number of addresses we can support and limit the size of problems. We are investigating alternatives, however creating an additional data structure for wire memory types that needs to be stored exacerbates the memory access problem. In the future we plan to also consider High Bandwidth Memory (HBM). Currently, AWS F1 instances do not support HBM.

We investigated two algorithm classes, traversal and threshold. In general, the more data that an algorithm places in ultraRAM, the fewer memory accesses required to run an application. For small problems where all the data fits in ultraRAM the algorithm and policy did not make a difference. However, when the amount of data processed by the application becomes very large, the approach used to assign data to memory becomes important. We are continuing to investigate larger applications. We investigated both traversal and thresholding algorithms. With the traversal approach, we expect that storing wires that will be used in the near future and with higher frequency should be best for larger and larger problems. Given that, for K-means, 80% of wires are used once or twice, storing these wires in ultraRAM should give the highest reuse of ultraRAM and hence the best performance. This behavior may change with different applications with different memory access patterns.

The thresholding approach met but did not out perform the traversal approach for large problems. There are several ways to improve thresholding, which we will explore, including to consider where in a layer a wire is used. There is a tradeoff

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

TABLE IV: traversal algorithm total number of wires stored in ultraRAM

# data points	# classes	p1	p2	p3	p4
100	2	1.34E+06	1.29E+06	1.32E+06	1.32E+06
100	4	2.66E+06	2.56E+06	2.61E+06	2.61E+06
100	8	5.30E+06	5.21E+06	5.21E+06	5.21E+06
1000	2	1.33E+07	1.28E+07	1.31E+07	1.31E+07
1000	4	2.64E+07	2.54E+07	2.59E+07	2.59E+07
1000	8	5.26E+07	5.07E+07	5.17E+07	5.17E+07

TABLE V: traversal algorithm memory access clock cycles

# data points	# classes	p1	p2	p3	p4
100	2	1.22E+07	1.58E+07	1.39E+07	1.39E+07
100	4	2.40E+07	3.12E+07	2.75E+07	2.75E+07
100	8	4.75E+07	6.20E+07	5.45E+07	5.45E+07
1000	2	1.21E+08	1.56E+08	1.38E+08	1.38E+08
1000	4	2.83E+08	3.41E+08	3.04E+08	3.04E+08
1000	8	1.16E+09	1.07E+09	1.02E+09	1.02E+09

TABLE VI: threshold algorithm number of wires stored in ultraRAM and memory access clock cycles

# data	# classes	2S (# of wires)	2S (clock cycles)	8S (# of wires)	8S (clock cycles)	first 1/4 (# of wires)	first 1/4 (clock cycles)
100	2	1.34E+06	1.22E+07	1.34E+06	1.22E+07	1.05E+06	4.48E+07
100	4	2.66E+06	2.40E+07	2.66E+06	2.40E+07	2.09E+06	8.87E+07
100	8	4.16E+06	1.77E+08	5.30E+06	4.75E+07	4.16E+06	1.77E+08
1000	2	1.04E+07	4.44E+08	1.33E+07	1.21E+08	1.04E+07	4.44E+08
1000	4	4.36E+06	2.27E+09	2.07E+07	8.79E+08	2.07E+07	8.79E+08
1000	8	8.71E+06	4.52E+09	4.13E+07	1.75E+09	4.13E+07	1.75E+09

TABLE VII: Best Memory Access Clock Cycles Comparison

# data	# classes	# wires stored in ultraRAM	best traversal clock cycles	# wires stored in ultraRAM	best threshold clock cycles	lower bound	upper bound
100	2	1.34E+06	1.22E+07	1.34E+06	1.22E+07	8.06E+06	1.42E+08
100	4	2.66E+06	2.40E+07	2.66E+06	2.40E+07	1.60E+07	2.83E+08
100	8	5.30E+06	4.75E+07	5.30E+06	4.75E+07	3.20E+07	5.65E+08
1000	2	1.33E+07	1.21E+08	1.33E+07	1.21E+08	7.99E+07	1.33E+09
1000	4	2.64E+07	2.83E+08	2.07E+07	8.79E+08	1.59E+08	2.64E+09
1000	8	5.17E+07	1.02E+09	4.13E+07	1.75E+09	3.17E+08	5.59E+09

in thresholding regarding the amount of time assigning a wire location and the efficiency of the result. Our goal is to have an algorithm that is both efficient to run and produces high quality results.

V. Conclusions and Future Work

We have investigated using ultraRAM on AWS F1 instances to accelerate processing of applications with big data, specifically those that can be abstracted as a directed graph. Our results clearly show a large advantage obtained using ultraRAM. As problem sizes grow and the amount of data stored in DDR increases, the policy used to choose between what goes in ultraRAM and what goes in DDR has increasing importance regarding overall application speed. Our results show that choosing the policy that best matches the application is important. For large amounts of data, keeping data with many accesses and data with short lifetimes in ultraRAM gives the best results.

Future work includes applying the best policy to a complete, implementation of garbled circuits on an application with a large amount of data, and showing the speedup achieved. In addition, we plan to investigate using HBM in our design. A more complicated memory allocation to accommodate specifics of the hardware design will need to be exploited. There is always a tradeoff between sophistication of the policy and run time devoted to preprocessing, which we will continue to explore.

Acknowledgements

The authors gratefully acknowledge the support of the National Science Foundation (grant CNS-1717213), Amazon Web Services, and Xilinx.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

REFERENCES

[1] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Taipei, Taiwan: IEEE, Oct. 2016, pp. 1–13. [Online]. Available: http://ieeexplore.ieee.org/document/7783710/

[2] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*. Monterey, California, USA: ACM Press, 2017, pp. 237–246. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3020078.3021742

[3] J. Sheng, C. Yang, A. Sanaullah, M. Papamichael, A. Caulfield, and M. C. Herbordt, "HPC on FPGA clouds: 3D FFTs and implications for molecular dynamics," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Ghent, Belgium: IEEE, Sep. 2017, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/8056853/

[4] M. Leeser, M. Gungor, K. Huang, and S. Ioannidis, "Accelerating Large Garbled Circuits on an FPGA-enabled Cloud," in *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. Denver, CO, USA: IEEE, Nov. 2019, pp. 19–25. [Online]. Available: https://ieeexplore.ieee.org/document/8945639/

[5] Xilinx, "UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices," Jun. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[6] Amazon, "Enable faster FPGA accelerator development and deployment in the cloud." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[7] Xillinx, "Ultraram: Breakthrough embedded memory integration on ultrascale+ devices." [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[8] M. Asiatici and P. Ienne, "DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. Barcelona, Spain: IEEE, Sep. 2019, pp. 254–262. [Online]. Available: https://ieeexplore.ieee.org/document/8892073/

[9] Z. Shao, R. Li, D. Hu, X. Liao, and H. Jin, "Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside CA USA: ACM, Feb. 2019, pp. 320–329. [Online]. Available: https://dl.acm.org/doi/10.1145/3289602.3293900

[10] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High Bandwidth Memory on FPGAs: A Data Analytics Perspective," *arXiv:2004.01635 [cs]*, Apr. 2020, arXiv: 2004.01635. [Online]. Available: http://arxiv.org/abs/2004.01635

[11] 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)*. Waltham, MA, USA: IEEE, Sep. 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8916407/

978-1-7281-9219-2/20/$31.00 ©2020 IEEE