Optimizing FPGA Memory Allocation for Matrix-Matrix Multiplication using Bayesian Optimization

Mehmet Gungor
Department of ECE
Northeastern University
Boston MA, USA
0000-0001-5154-1809

Stratis Ioannidis
Department of ECE
Northeastern University
Boston MA, USA
0000-0001-8355-4751

Miriam Leeser
Department of ECE
Northeastern University
Boston MA, USA
0000-0002-5624-056X

***Abstract*—Matrix-matrix multiplication (MM) of large matrices plays a crucial role in various applications, including machine learning. MM requires significant computational resources, but accessing memory can quickly become the bottleneck. Field-Programmable Gate Arrays (FPGAs) offer a range of memory options, such as Block RAM (BRAM), UltraRAM (URAM), and High Bandwidth Memory (HBM), each with unique characteristics. In this study, we explore the optimal combination of HBM with either BRAM, URAM, or both, depending on the size of the input data. We employ Bayesian optimization to optimize the FPGA implementation, analyze the trade-offs between different memory types, and determine the most suitable memory allocation based on memory sizes. Our findings provide insights for designers seeking to optimize their designs, and demonstrate that URAM outperforms a combination of BRAM and URAM when data fits in URAM. Overall, our approach enables more efficient memory allocation for larger matrix sizes on FPGAs compared to prior research.**

***Index Terms*—matrix-matrix multiply, FPGA, High Bandwidth Memory, Bayesian Optimization**

I. Introduction

Matrix-matrix multiplication is an important component in many modern applications, including networking applications and machine learning. With big data, multiplying large matrices is becoming increasingly important. While matrix matrix multiplication is compute bound for small matrices, it quickly becomes memory bound for larger applications.

As data access has increasingly become the bottleneck, accelerator manufacturers have integrated many different types of memory to improve data access. As its name implies, High Bandwidth Memory (HBM) provides higher bandwidth memory compared to DDR and is widely used in machine learning applications. NVIDIA introduced HBM into its GPUs in 2016, and Xilinx introduced HBM in its Ultrascale+ FPGAs in 2016 [1]. The same Ultrascale+ devices also introduced UltraRAM (URAM) [2]. There is more URAM on an Ultrascale+ FPGA, but it supports fewer ports than Block RAM (BRAM).

While these different types of memory are provided to help users achieve the best data access for their applications, there are few optimization tools and little guidance regarding how best to use them. We address this gap in this paper. Specifically, we investigate the use of different types of memory on an FPGA with the goal of determining the best assignment of data to memory for fast Matrix multiplication. As an application we analyze GEMM: General matrix multiply, one of the Basic Linear Algebra subprograms, and investigate how best to optimize memory assignments to achieve good performance. We start with data being transferred from the host to the HBM. We then investigate how best to bring blocks of memory into either BRAM or URAM to efficiently feed data blocks for processing on the FPGA fabric and identify the best assignment of data to memory to improve throughput and system performance.

We apply Bayesian optimization to choose the best memory allocation. Several studies discuss how FPGAs have been used to accelerate Bayesian optimization. There are few examples where Bayesian optimization is applied to the hardware design. This paper is the first application of Bayesian optimization to memory allocation. We use Pareto graphs to provide guidance for selecting the best memory allocation, based on the size of the input matrices.

The contributions of this paper are:

- A quantitative analysis of the allocation of different types of memory on an FPGA to improve overall performance.
- The application of Bayesian optimization to determine the best allocation of data to different memory types on an FPGA.
- Advice to designers regarding how to allocate memory based on the results of the Bayesian optimizer and an analysis of the Pareto frontier.

The remainder of the paper is organized as follows. We present background on matrix multiplication, different memory types, Bayesian optimization and related work in Sec. II. We present our methodology and experiments in Sec. III, and present and discuss results in Sec. IV. We end with conclusions and a discussion of future work. More details can be found in the first author's PhD dissertation [3].

II. Background

A. Matrix Matrix Multiplication

Matrix multiplication is a widely used application in machine learning including linear regression and principal component analysis. Neural Networks (NNs) rely heavily on matrix operations. In this research we examine GEMM: GEneral Matrix Multiplication , one of the Basic Linear Algebra Subprograms (BLAS), which involves multiplying matrices A and B. Optimizations of GEMM frequently involve decomposing one or both of A, B into block matrices. We use block matrix multiplication in this work. Our implementation is straightforward. Our focus is not on the most efficient MM implementation, but rather on memory access patterns.

B. FPGA Memories

TABLE I
Memory for Alveo U280

Memory	Capacity	Bandwidth	Ports	Rd Latency
Block RAM	9.072MB	5.4GB/s	2 per BRAM	1-2
UltraRAM	34.56MB	1.35GB/s	2 per block	1-5
HBM	8GB	460GB/s	32 (max)	40-50
DDR	32GB	38 GB/s	2	50-60

There are several different types of memory available on the Ultrascale+ devices from AMD. We focus on the AMD Alveo line of data center accelerator cards with High Bandwidth Memory [4]. We examine the use of HBM in conjunction with other memory types and how the memory allocation affects the performance of GEMM. The experiments apply directly to Alveo U50, Alveo U55c and Alveo U280 accelerator cards. We did our experiments on the U280, which has DRAM; however, DRAM was not used in this research. Our approach examines the use of Block RAM (BRAM) and UltraRAM (URAM) in conjunction with High Bandwidth Memory (HBM). The amount of different types of memory on the U280 and their characteristics is shown in Table I. Specifically, the table shows total capacity, peak bandwidth, maximum ports, and read latency in clock cycles. Each Block RAM in the Xilinx UltraScale architecture-based devices stores up to 36 Kbits of data and can be configured as either two independent 18 Kb RAMs, or one 36 Kb RAM. Block RAM (BRAM) is integrated within the FPGA fabric. The Alveo U280 has 2016 Block RAMs which can be configured as true dual port or simple dual port. True dual port BRAMs have two independent ports that can be used for either writing or reading. In contrast, when the block RAM is used in simple dual port mode, independent read and write operations can occur simultaneously, however port A is designated as the write port and port B as the read port. Hence simple dual port mode is less flexible than true dual port mode.

In addition to BRAM, the Alveo U280 also has 960 UltraRAMs on chip. UltraRAM blocks are 288 Kb, organized as 72 bits x 4K entries. URAMs are single clock, synchronous memory blocks arranged in columns on the device. Each UltraRAM has two ports, and each port can perform either a read or a write operation per cycle. UltraRAM should provide fast access as it is on chip.

HBM is an in package, but off-chip memory. Due to the way the memory is integrated with the FPGA, high bandwidth can be supported. The Alveo U280 has 8GB of HBM with peak memory bandwidth of 460GB/sec. For HBM, 32 pseudo channels connect to 16 physical channels. However, there may be routing complexity added when they are all connected to the design. The peak bandwidth happens when all the ports are reading data at the same time from different banks in burst mode. We benchmarked the case where they are accessing data from the same bank. If we let 8 or 16 ports issue 25 read operations to the same memory bank simultaneously and the ports do not operate in burst mode, the latency for each port is observed to be 40-50 clock cycles.

C. Bayesian Optimization

Bayesian optimization [5] is an optimization framework designed to optimize objective functions that are expensive to evaluate. It is particularly well-suited for global optimization of black-box functions – those for which the analytical form is unknown and derivatives are not available, but function calls themselves are expensive to compute; a canonical example is hyperparameter tuning in deep learning [5]. On a high-level, the method relies on Bayesian inference to model the objective function, and produces a sequence of inputs that both explore the input space, but also are in regions where the objective is indeed optimized. Formally, Bayesian optimization models the function as a Gaussian process [6] with a known covariance (usually determined by a kernel, such as Radial Basis Function or RBF). As observations (evaluations of the objective function) are collected, the Gaussian process assumption allows computing a posterior over unseen function values. The posterior can be used to both (a) predict function values for new inputs in expectation, thereby interpolating between values already seen, but also (b) measure the uncertainty in these predictions.

Bayesian optimization leverages this to select which inputs to sample (i.e., evaluate the function on) next. In particular, inputs to be explored are determined by picking points that maximize a so-called acquisition function: this usually takes into account the expected value but also the uncertainty at that location. Using for example the upper-confidence bound (UCB) acquisition function, exploration can trade-off between sampling parts of the input space where the function is expected to be large, while also exploring inputs where the function is highly variant. Thus, the sequence of evaluations produced is guided towards finding an optimal value, while not leaving regions of the input space unexplored.

We use a publicly available Bayesian optimizer [7] to explore the best allocation of data to memory types. We use a Gaussian/RBF kernel with a Upper COnfidence Bound (UCB) acquisition function. The black box function runs the make file for the HLS code as described in Sec. III. This is a good match for Bayesizan optimization, as the equation for implementing

the optimization is unknown and highly non-linear and the evaluation of each design point takes hours.

D. Related Work

There are several papers describing the results of mapping applications to FPGAs that take advantage of different types of memory. ScalaBFS [8] runs the BFS algorithm on multiple processing elements and uses BRAM and URAM to store intermediate states. There is no discussion presented regarding the size or type of URAM vs. BRAM, and their achieved clock speed, 90MHz, is low. The low frequency may be due to routing to memory.

A number of papers highlight how HBM can benefit an application. [9] shows how HBM can benefit a database application. Graphlily [10] proposes an overlay to accelerate GraphBLAS on an FPGA with HBM. [11] analyzes the use of HBM by getting benchmarks of real world usage of HBM channels and using different access patterns, resulting in advice to increase the performance of HBM.

Several researchers have looked at tools for optimizing memory bandwidth and latency. HBM Connect [12] is a tool for optimizing the bandwidth for HBM accesses. The Dynaburst project [13], [14] looks at reordering accesses to DRAM for non-streaming applications to improve DDR access times. While these tools are solving similar problems to the one presented here, they do not consider HBM in conjunction with tradeoffs between BRAM and UltraRAM. An older paper [15] looks at optimizing different on-chip memory resources on an FPGA, but the paper predates the availability of HBM.

[16] focuses on restructuring BRAM resources on an FPGA by inferring memory allocation in HLS by providing three different models and an automation tool that analyzes performance and resource trade-offs to optimize system performance. This research focuses on BRAM, while our research includes BRAM, URAM and HBM, thus covering a larger design space on FPGA hardware.

Others have used FPGAs to accelerate Bayesian optimization [17]–[19]. We use Bayesian optimization to improve our FPGA design, specifically to choose the best memory allocation for a particular matrix multiply problem. Others have applied Bayesian optimization to circuit design [20], [21]. A recent paper applies Bayesian optimization to optimizing the resources on an FPGA from an HLS implementation [22]. This is the first time that Bayesian optimization has been applied to memory allocation on FPGAs.

III. Methodology

We have selected a block matrix multiplication design that takes two large dense square matrices from HBM in smaller square blocks, and processes these blocks on the FPGA. Our design assumes the input and output matrices fit completely in HBM. For the Alveo U280, this means we can multiply two 2.6 GB input matrices. For our current results, the largest size we test is square matrices with close to 18 Megabytes for each input matrix. The size is limited by the time it takes to create and optimize the design, and not by the implementation. Larger matrices can easily be handled with the underlying block-based algorithm. Matrices where the inputs and output do not entirely fit into HBM can also be handled if they are broken up into blocks by the host and fed to the FPGA memory.

The block matrix matrix multiplication design is implemented with Vitis HLS. We place square matrices in HBM from the host. Our design separates each input matrix into multiple square blocks and reads each block from a separate channel to HBM. These block are multiplied in parallel; for an MxM block size the parallelization is M. After execution, the result is added to an output cache, which holds the final result, and is then written back to off-chip memory. For lock matrix multiplication we unroll the innermost for loop in order to do MAC operations in parallel, and we partition the block arrays accordingly. The implementation cycles through until all blocks are multiplied and placed back in HBM. We read each row of the input matrix block from HBM to on-chip memory (BRAM, URAM, or a combination) so that the matrix multiplication computation can be done efficiently by reading input data from on-chip memory. We do design space exploration in HLS in terms of resource utilization and timing effects that result when using different sizes and types of on-chip memories.

There are many optimizations that can be applied, even to our straightforward MM implementation, in High Level Synthesis (HLS). We apply several HLS pragmas, including array partitioning. The first input is partitioned by row and the second input array is partitioned by its columns to multiply rows and columns in parallel. We usethe bind storage pragma to assign on-chip memories for inputs, outputs and the output cache. We also apply unrolling to the inner loop of block matrix matrix multiplication.

We explore how to map the blocks to on-chip memory, and consider Block RAMs, URAMs and a combination. We ignore the fact that CLBs can also be used as memory, because we want the design to be as resource efficient as possible, and on-chip dedicated memory is a more efficient use of resources. As the size of on-chip memory required increases, the routing between memory elements can become an issue. This results in a lower clock frequency in the design in order to meet timing requirements.

To explore the memory choices we use a Bayesian optimizer [7]. In our case, to evaluate the objective function, we generate and verify the new generated design, calculate performance and store generation reports in each iteration; this step can take 2 to 3 hours.

The overall optimization process works as follows. We start with the HLS code for block matrix-matrix multiplication, where block size is a parameter. We give the Bayesian optimizer a range of block sizes that can fit in on-chip memories and we set the on-chip memory type to try along with the HLS code. The optimization goal is throughput of the design. Throughput is measured from the host, and defined as the number of bytes in the input matrices divided by the runtime.

The optimizer chooses a block size within the input range, runs HLS and generates a xclbin file and host code that can verify and measure the throughput of the design. The generated bitstream is then downloaded to the Alveo U280 and run. The throughput is measured and returned to the optimizer. This process is automated with a python script where a user only needs to provide memory ranges and memory type.

5Define throughput: Given size of Input matrices in bytes, measure the time from host until result is returned to the host. throughput = bytes/time.

In order to download the design generated by Vitis HLS to the board, we link the IP with a shell from AMD [23]. Both the generated IP block and the host code changes with block size. The Bayesian optimizer, given a block size range and on-chip memory type does the following:

1) change the HLS code block size parameter and sets on-chip memory types as the user provided.
2) changes the host code test matrix size.
3) builds the project to generate the xclbin file
4) builds the host code
5) loads the xclbin file to the board and runs the host binary
6) reads the performance results and returned it to the Bayesian optimizer.
7) Bayesian optimizer selects a new block size according to the performance and goes back to step 1.

We explore with 5 iterations for each memory type and in addition we experiment with the URAM design in 3 different ranges to compare with other memory options.

A. Experiments

For our experiments we use an AMD/Xilinx Alveo U280 card that has 7.8 MB of on-chip Block Memory (BRAM) and 33 MB of Ultra RAM (URAM) as well as 8 GB of HBM with 32 pseudo channels. For High Level Synthesis we use Vitis 2022.1. Each experiment requires 2 to 3 hours to generate a solution. We generate two random matrices on the host machine and transfer them to the FPGA's HBM. We experimented with matrices in the range of .5 Megabytes (200 x 200 elements) to close to 18 Megabytes (2164 x 2164 elements). Then we start our matrix matrix multiplication design, which runs until all blocks are multiplied and the results are stored back in HBM. Finally the results are read back to the host. We measure end to end performance starting with the time that the input matrices are sent to the FPGA and ending with the read back of the results from the FPGA by the host, so total time includes matrix transfer time between host and FPGA. We integrate the Bayesian optimizer by giving it the configuration of memories and the block matrix upper and lower size bounds, The Bayesian optimizer tries different block matrix sizes, gets performance results, and tries to optimize the overall performance. A range of block matrix sizes are given to the Bayesian optimizer as an input parameter and the output parameter is the measured throughput of that particular design.

IV. Results

Our results examine the mapping of many different designs onto the Alveo U280. Each design represents a different size of matrix and a different assignment of blocks to memory. We experiment with different ranges, for BRAM experiments we set each block is between 100x100 to 600x600, for URAM designs we set block sizes 100x100 to 900x900, and for the combination design we map two inputs on URAM and the output cache to BRAM and set ranges from 400x400 to 1200x1200. Our goal was to push the size of data needed in our designs.

In Fig. 1 we show the overall clock frequency of designs for different sizes of matrix blocks and different assignment of these blocks to memory as a Pareto graph. In Fig. 2 we show the same experiments with the figure of merit being overall throughput. Designs that use only BRAM are labeled hbx, designs that use only URAM are labeled huX, and designs that use a combination are labeled hcx.

We observed that clock frequency between different designs changes significantly even for very similar block sizes. This is largely due to routing complexity. While no clear rule emerges for assigning memory types, we do observe important trends. For smaller sizes of matrices, where the blocks fit completely into BRAM, many designs have high clock frequency and high throughput, and BRAM can be a good choice. The clock speed for designs that use only BRAM drops significantly when more than 50% of BRAM is used, for example in hb6.

URAM is competitive for small block sizes. Since BRAMs are smaller, you need more of them which increases the routing complexity. Since we are only using one read port and one write port for our designs, assigning data to URAM provides very good results even at small block sizes. Some of the best performing designs, hu17 and hu18, use only URAM, or use a very small amount of BRAM for other purposes.

We experimented with mapping input matrices to both BRAM and URAM with a 1:2 ratio in order to can fit larger matrices when BRAM alone does not provide enough storage. Designs that use a combination of URAM and BRAM always perform worse than only using URAM. This combination should only be chosen if there is insufficient URAM to store all the data. This is due to the increased routing congestion and can be seen in all of the hcx designs.

Based on these experiments, we observe that there is not an easy rule for choosing the optimal mapping of data to memory, other than to avoid a combination of memories. Bayesian optimization is a good choice for a designer wishing to optimize their designs as it automates the search of the design space.

In previous work, the authors [24] do not consider using URAM and only using BRAM which leaves a big portion of on-chip memory un-utilized. We believe their designs could be improved by making use of URAM on the chip.

There has been much research conducted on optimizing the matrix matrix multiplication kernel, and we plan to continue to experiment with these techniques. We experimented with

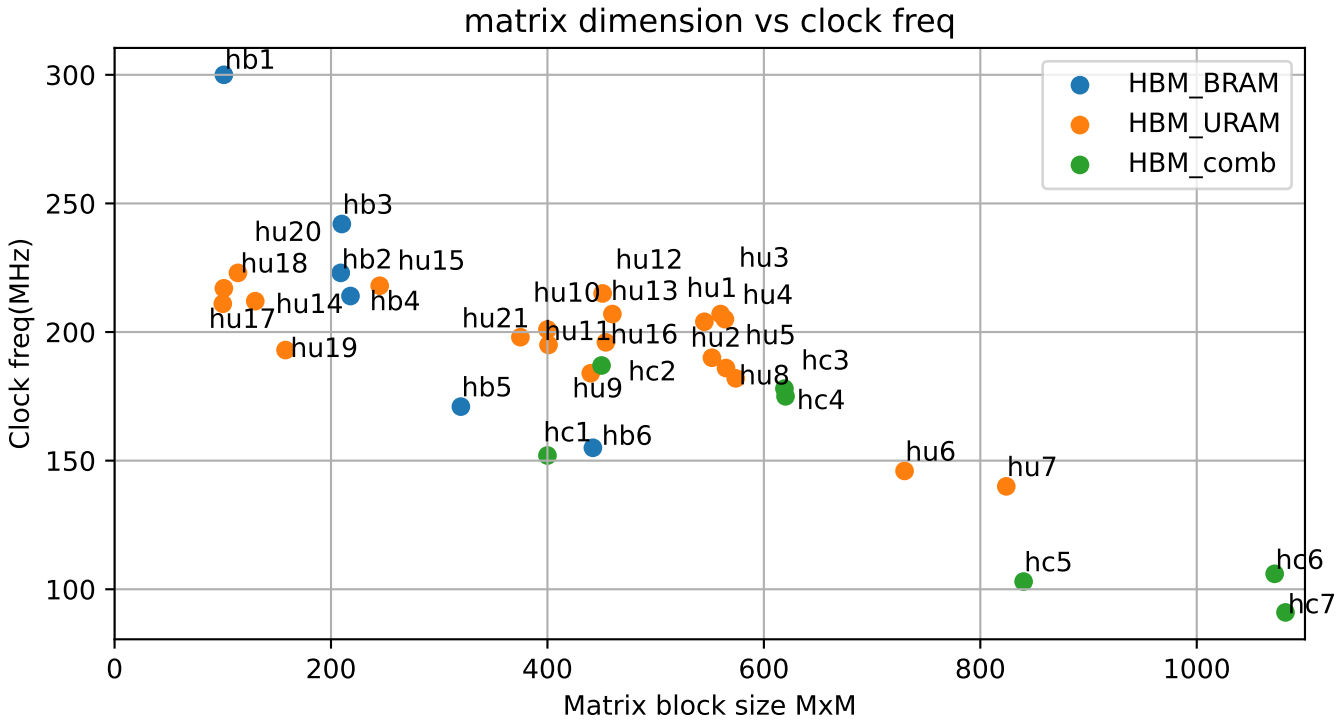

Fig. 1. Matrix sizes vs clock frequencies

making our design more parallel. However, highly parallel designs run out of DSP resources available on the chip and can not utilize all the on-chip memories available. [25] propose a sparse dense matrix matrix (SpMM) multiplication method. They reschedule to access non zero elements of a matrix and utilizing on-chip BRAM memory and HBM channels on an Alveo U280. Their design utilizes 76% percent of total available BRAM and reaches peak performance of 181.1 GFLOP/s with a clock speed of 189MHz. Their flexible design can be reused for different hardware. We plan to investigate this and other advanced MM implementations in the future.

V. Conclusions and Future Work

Matrix matirx multiplication is a widely used part of many applications. It has become increasingly important in machine learning, where large sizes of matrices are processed. There are numerous different ways to optimize the matrix matrix multiplication kernel. In this research, we focus on optimizing the memory usage and the ability to feed data to the processing. We use a combination of high bandwidth memory and on-chip memory (BRAM and URAM) to determine the best assignment of data to memory to achieve fast clock cycles and good throughput. This work represents one of the few applications of Bayesian optimization to FPGA design, and is the first to use Bayesian optimization for memory allocation on an FPGA. Our results show that Bayesian optimization is a good match for this problem whose results are highly non-linear.

There is much work that has been done in optimizing the matrix matrix multiplication algorithm itself. In the future we plan to combine kernel optimization with memory optimization to further improve GEMM performance. We will investigate applying Bayesian optimization to the MM kernel. We will also investigate applying our technique to other applications to improve the use of memory on the FPGA. We also plan to investigate optimizations targeting the Versal architecture.

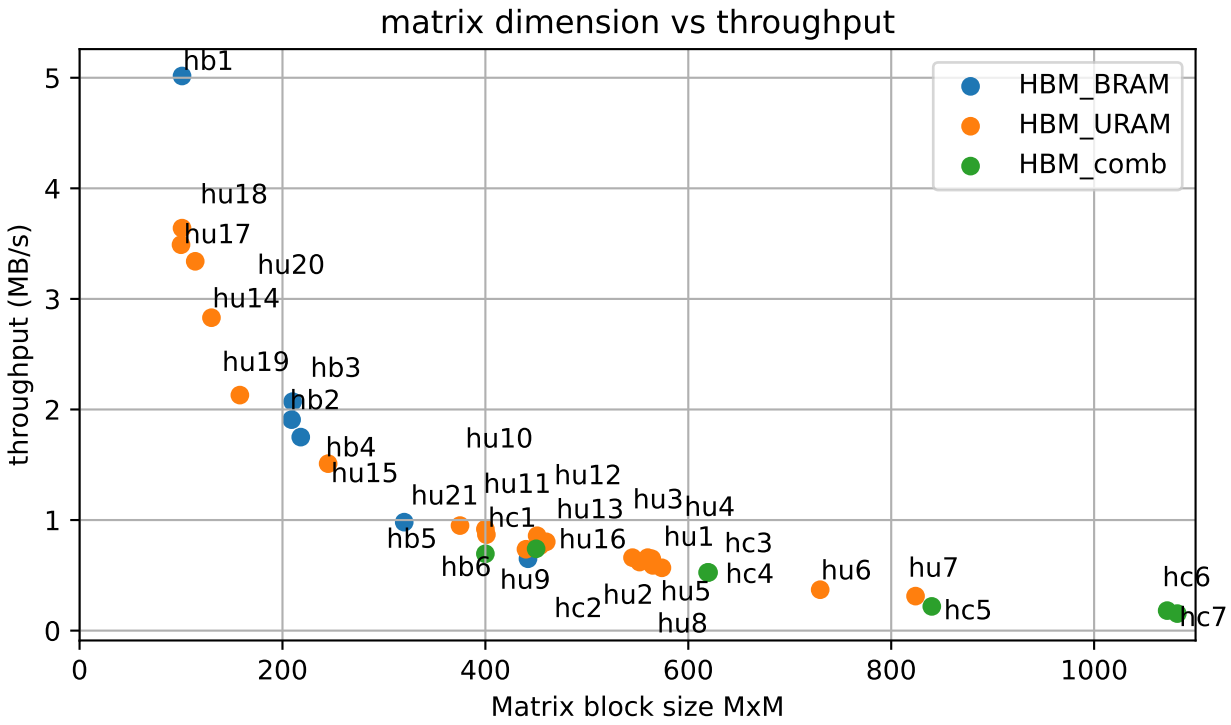

Fig. 2. Matrix sizes vs throughput

TABLE II
RESULTS FOR DIFFERENT EXPERIMENTS SORTED BY MATRIX SIZE

Experiment ID	Experiment type	Block size MxM	Matrix size (MB)	Clock freq(MHz)	BRAM utilization(%)	URAM tilization(%)	throughput (MB/s)
hu17	HBM with URAM	100x100	0.153	211	0.11	41.67	3.49
hb1	HBM with BRAM	101x101	0.156	300	22.4	0	5.016
hu18	HBM with URAM	101x101	0.156	217	0.11	21.25	3.64
hu20	HBM with URAM	114x114	0.198	223	0.11	23.75	3.34
hu14	HBM with URAM	130x130	0.258	212	0.11	27.08	2.83
hu19	HBM with URAM	158x158	0.381	193	0.11	32.92	2.13
hb2	HBM with BRAM	209x209	0.667	223	35.81	0	1.908
hb3	HBM with BRAM	210x210	0.673	242	35.94	0	2.071
hb4	HBM with BRAM	218x218	0.725	214	36.93	0	1.75
hu15	HBM with URAM	245x245	0.916	218	0.11	34.17	1.51
hb5	HBM with BRAM	320x320	1.563	171	49.58	0	0.981
hu21	HBM with URAM	375x375	2.146	198	0.11	39.17	0.95
hc1	HBM with comb	400x400	2.441	152	19	62	0.695
hu10	HBM with URAM	400x400	2.441	201	0.11	41.67	0.918
hu11	HBM with URAM	401x401	2.454	195	0.11	33.75	0.868
hu9	HBM with URAM	440x440	2.954	184	0.11	36.67	0.736
hb6	HBM with BRAM	442x442	2.981	155	64.71	0	0.65
hc2	HBM with comb	450x450	3.090	187	24.7	46.88	0.74
hu12	HBM with URAM	451x451	3.104	215	0.11	37.92	0.858
hu16	HBM with URAM	454x454	3.145	196	0.11	37.92	0.77
hu13	HBM with URAM	460x460	3.229	207	0.11	38.33	0.803
hu1	HBM with URAM	545x545	4.532	204	0	37.9	0.66
hu2	HBM with URAM	552x552	4.649	190	0	38.33	0.62
hu3	HBM with URAM	560x560	4.785	207	0	39.17	0.66
hu4	HBM with URAM	564x564	4.854	205	0	39.17	0.65
hu5	HBM with URAM	565x565	4.871	186	0	39.58	0.59
hu8	HBM with URAM	574x574	5.027	182	0.11	40	0.568
hc3	HBM with comb	619x619	5.847	178	40.6	48.44	0.526
hc4	HBM with comb	620x620	5.865	175	40.6	48.44	0.526
hu6	HBM with URAM	730x730	8.131	146	0	76.67	0.37
hu7	HBM with URAM	824x824	10.360	140	0.11	76.67	0.312
hc5	HBM with comb	840x840	10.767	103	83.48	100	0.219
hc6	HBM with comb	1072x1072	17.535	106	86.33	91.88	0.181
hc7	HBM with comb	1082x1082	17.864	91	88.02	92.86	0.154

ACKNOWLEDGMENT

This research was partially funded by grant NSF SATC 1717213. The authors would like to thank colleagues at Northeastern University for useful conversations, especially members of the Reconfigurable Computing Laboratory.

REFERENCES

[1] "Virtex UltraScale+ HBM Devices," 2016. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-hbm.html

[2] "UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices," Jun. 2016. [Online]. Available: https://docs.amd.com/v/u/en-US/wp477-ultraram

[3] M. Gungor, "Optimizing the use of different memory types on modern fpgas," Ph.D. dissertation, Northeastern University, Boston, MA, 2024.

[4] "UltraScale Architecture Memory Resources User Guide," September 2021. [Online]. Available: https://docs.amd.com/v/u/en-US/ug573-ultrascale-memory-resources

[5] P. I. Frazier, "A tutorial on bayesian optimization," *arXiv preprint arXiv:1807.02811*, 2018.

[6] E. Schulz, M. Speekenbrink, and A. Krause, "A tutorial on gaussian process regression:Modelling, exploring, and exploiting functions," *Journal of Mathematical Psychology*, vol. 85, pp. 1–16, 2018.

[7] F. Nogueira, "Bayesian Optimization: Open source constrained global optimization tool for Python," 2014–. [Online]. Available: https://github.com/bayesian-optimization/BayesianOptimization

[8] C. Liu, Z. Shao, K. Li, M. Wu, J. Chen, R. Li, X. Liao, and H. Jin, "Scalabfs: A scalable bfs accelerator on fpga-hbm platform," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 147–147.

[9] 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

[10] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[11] P. Holzinger, D. Reiser, T. Hahn, and M. Reichenbach, "Fast hbm access with fpgas: Analysis, architectures, and applications," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 152–159.

[12] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: High-performance hls interconnect for fpga hbm," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 116–126.

[13] 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/

[14] G. Csordas, M. Asiatici, and P. Ienne, "In Search of Lost Bandwidth: Extensive Reordering of DRAM Accesses on FPGA," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. Tianjin, China: IEEE, Dec. 2019, pp. 188–196. [Online]. Available: https://ieeexplore.ieee.org/document/8977899/

[15] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein, and J. Emer, "Scavenger: Automating the construction of application-optimized memory hierarchies," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–8.

[16] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for hls," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[17] H. Fan, M. Ferianc, M. Rodrigues, H. Zhou, X. Niu, and W. Luk, "High-performance fpga-based accelerator for bayesian neural networks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1063–1068.

[18] H. Fan, M. Ferianc, Z. Que, S. Liu, X. Niu, M. R. Rodrigues, and W. Luk, "Fpga-based acceleration for bayesian convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 12, pp. 5343–5356, 2022.

[19] M. Ferianc, Z. Que, H. Fan, W. Luk, and M. Rodrigues, "Optimizing bayesian recurrent neural networks on an fpga-based accelerator," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021, pp. 1–10.

[20] H. M. Torun, M. Swaminathan, A. K. Davis, and M. L. F. Bellaredj, "A global bayesian optimization algorithm and its application to integrated system design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 4, pp. 792–802, 2018.

[21] S. Zhang, F. Yang, C. Yan, D. Zhou, and X. Zeng, "An efficient batch-constrained bayesian optimization approach for analog circuit synthesis via multiobjective acquisition ensemble," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 1–14, 2021.

[22] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, "Bayesian optimization for efficient accelerator synthesis," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1145/3427377

[23] "U280 Gen3x16 XDMA base_1 Platform," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1120-alveo-platforms/U280-Gen3x16-XDMA-base_1-Platform

[24] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible communication avoiding matrix multiplication on fpga with high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 244–254.

[25] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 65–77.