A Scalable Architecture for CNN Accelerators Leveraging High-Performance Memories

Maarten Hattink*, Giuseppe Di Guglielmo†, Luca P. Carloni†, and Keren Bergman*
* *Department of Electrical Engineering,* † *Department of Computer Science*
Columbia University in the City of New York, New York, USA
mh3654@columbia.edu, giuseppe@cs.columbia.edu, luca@cs.columbia.edu, bergman@ee.columbia.edu

***Abstract*—As FPGA-based accelerators become ubiquitous and more powerful, the demand for integration with High-Performance Memory (HPM) grows. Although HPMs offer a much greater bandwidth than standard DDR4 DRAM, they introduce new design challenges such as increased latency and higher bandwidth mismatch between memory and FPGA cores. This paper presents a scalable architecture for convolutional neural network accelerators conceived specifically to address these challenges and make full use of the memory's high bandwidth. The accelerator, which was designed using high-level synthesis, is highly configurable. The intrinsic parallelism of its architecture allows near-perfect scaling up to saturating the available memory bandwidth.**

I. Introduction

Using Field Programmable Gate Arrays (FPGAs) to accelerate neural networks has gained both industrial and academic interest [1]–[4]. As the classes and applications of neural networks continues to grow both in size and variety, designers can tailor the FPGA reconfigurable hardware to meet their high computational demands. In particular, convolutional neural networks (CNNs) are widely adopted for a variety of applications, such as image and speech recognition.

Prior works have shown that maximizing the performance of CNNs on FPGAs requires using all their resources and, specifically, employ each available DSP to perform a multiplication almost every cycle [5], [6]. When the number of available DSPs in a FPGA becomes a performance bottleneck, one way to further increase the data-processing throughput is to apply a Winograd Transform [7]. By trading off a reduced number of multiplications for extra addition operations, the Winograd Transform delivers higher throughput with fewer DSPs. This optimization was demonstrated by Huang et al [8], who observe how the performance becomes limited by the available memory bandwidth before all DSPs in the FPGA can be put to use. This observation suggests that there is room for higher performance, if only more bandwidth was available.

Standard DRAM modules on high-end FPGA devices have a maximum bandwidth on the order of 20GB/s. For example, Xilinx's latest Ultrascale+ architecture delivers 21GB/s when operated with a 64-bit wide interface at 2666 MT/s [9]. Newer *high-performance memory (HPM)* technologies, based on stacked DRAM, allow for much higher bandwidth. A single Hybrid Memory Cube (HMC) allows up to 160GB/s to be connected to a FPGA through the FPGA's high speed transceivers [10]. High Bandwidth Memory (HBM), a different HPM technology that has been originally used for GPU-based systems, is available for FPGA boards as well. For example, it can achieve 460GB/s on Xilinx FPGAs when connected through a silicon interposer [11] . The goal of this paper is to investigate the HPM opportunity to realize more efficient accelerators for CNNs with FPGAs.

To the best of our knowledge, we present the first scalable architecture to accelerate CNN computation on FPGA systems that leverage HPM technology. We used the Winograd Transform [12] to raise the accelerator's throughput and address the computational bottleneck in conventional CNN architectures. We designed and implemented our CNN accelerator using a system-level design approach [13], i.e. combining a design specification made in C and using high-level synthesis for design-space exploration. In particular, high-level synthesis allowed us to explore the degree of spatial computation that we can apply by scaling the parallelism of our architecture to achieve high performance by maximizing the use of the available FPGA resources. Our architecture leverages the HPM bandwidth to address the increased access latency and bandwidth mismatch that occur when using this type of memory. We demonstrated the accelerator on a Micron AC-510 module [14], which contains a KCU060 FPGA connected to a 4GB HMC unit. We execute the VGG16 network [15], a CNN that has been studied extensively and used as an example to evaluate the performance of many other architectures in the literature. The experimental results show that the performance of our architecture scales in a near perfect way while increasing the utilization of FPGA resources up to saturation of the available HPM bandwidth.

II. Background

The VGG16 network is a deep CNN consisting of 13 convolution layers and 3 fully connected layers (FC). The layers process different numbers of input and output features: 224 (Layers 1-2), 112 (Layers 3-4), 56 (5-7), 28 (8-10), and 14 (11-13). Each input feature to a layer is convolved with a 3×3 filter. Each output feature is the sum of the convolution of all input features, with unique filters for each combination of input and output features. Feature size are reduced by applying a max-pool operation: each feature is divided in 2×2 grids, then from each grid the maximum value is taken and the rest discarded; the result is a new feature with half the width and height of the previous one. The FC layers are a matrix-vector

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

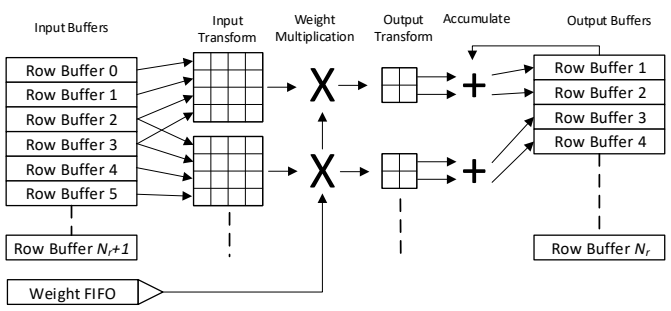

Fig. 1. Basic implementation of Winograd algorithm. Arrows indicate how input transforms overlap input data and how results are accumulated in the output buffer.

multiplication where the output of the previous layer is treated as a vector, which is multiplied with a matrix of weights; the result is a new vector which is the output of the layer.

III. Accelerator Architecture

A. Winograd Algorithm

The accelerator implements a convolutional neural network algorithm using the Winograd Transform [12]. This algorithm reduces the number of multiplications needed to compute a convolution at the expense of more addition operations. The accelerator implements a version of the Winograd Transform that has been explained in detail in [7], where it is denoted as $F(3 \times 3, 2 \times 2)$ and described by the following equation:

$$Y = A^T \Big[[GgG^T] \otimes [B^T dB] \Big] A \qquad (1)$$

The Winograd Transform takes a 4×4 matrix of input samples d, a 3×3 matrix of filter weights g, and linear transformations A, B, G, to compute a 2×2 matrix of output contributions Y. The linear transformations consist of additions and scaling by powers of 2, which are easy to implement with FPGAs. The transformations GgG^T and $B^T dB$ result in two 4×4 matrices that are multiplied element-wise (denoted with $\otimes$). Finally, the linear transformation A is applied to obtain a 2×2 matrix of output values Y. These output values are the same values as one would normally compute when doing a standard convolution, which requires 9 multiplications for each output value, or 36 multiplications for 4 output values. With the Winograd Transform, however, only 16 multiplications are required. The input transform costs an additional 32 additions, the filter transform 28 operations, and the output transform 24 additions. Because of the computational bottleneck in the number of available DSPs, this is a worthwhile tradeoff that allows the FPGA to achieve a higher throughput per DSP. Furthermore, we precompute the filters and store them in memory. This results in a higher memory-bandwidth requirement, but it reduces the number of operations on the FPGA. The values stored in memory are 16-bit fixed-point numbers, but the B transformation increases the size of input samples to 18 bits, which results (16×18)-bit multiplications. Those multiplications are implemented using a single DSP block. In order to preserve sufficient accuracy, the output feature

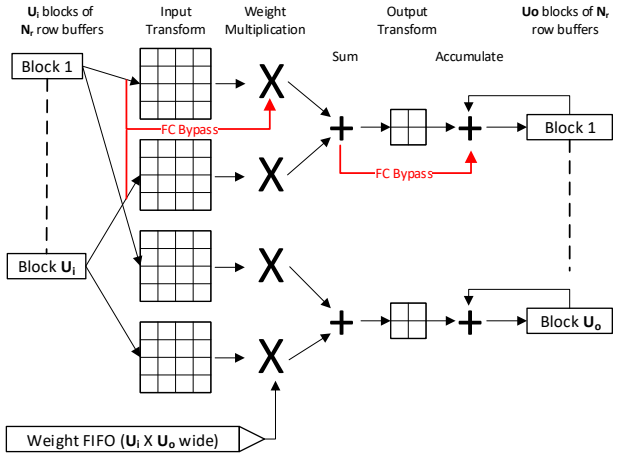

Fig. 2. Accelerator unrolling with $\boldsymbol{U_i}$=$\boldsymbol{U_o}$= 2. FC bypasses are marked in red, and only used during FC layers.

accumulation uses 32-bit fixed-point numbers. Before writing the results to memory, however, these are truncated to a 16-bit fixed-point format.

B. Compute Core Implementation

Fig. 1 shows the implementation of the Winograd algorithm by the accelerator. To increase reuse of loaded data, we compute multiple rows of the output features in parallel; the number of rows computed in parallel is denoted with $\boldsymbol{N_r}$. Each iteration of the algorithm requires four input samples per row buffer to compute two outputs per row. Two input samples are saved and reused in every cycle, but in different positions in the input matrix. Consequently, the number of read ports on the input-buffer memory is reduced. Since the algorithm requires two extra rows, there are $\boldsymbol{N_r}$+2 input row buffers needed to compute $\boldsymbol{N_r}$ output rows, and $\boldsymbol{N_r}$ output row buffers to store the results. From this 4×4 matrix of input data and the weights, we calculate the output contribution of the input feature. We load the accumulated value of the output feature from the output buffers, sum it with the new contribution, and store it again. Each row buffer can store the same set of rows for multiple features. First, the accelerator loops over all input features to calculate a single output feature. Then, it repeats this loop for the next output feature. To prevent the accelerator from having to store intermediate values of the output features to memory, and then loading them again, for a set of rows all input features are stored on the FPGA. This is done by instantiating $\boldsymbol{N_{bi}}$ blocks of input buffers, each consisting of $\boldsymbol{N_r}$+2 row buffers, that together can hold a set of rows for each input feature of the CNN layer currently processed by the accelerator.

C. Increase of Parallelism

To increase the accelerator's throughput, the core algorithm can be unrolled in two directions, as shown in Fig. 2. First, the converted input samples can be used to calculate the contribution for multiple output rows in parallel. The amount of output unrolling, denoted by $\boldsymbol{U_o}$, increases the number of

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

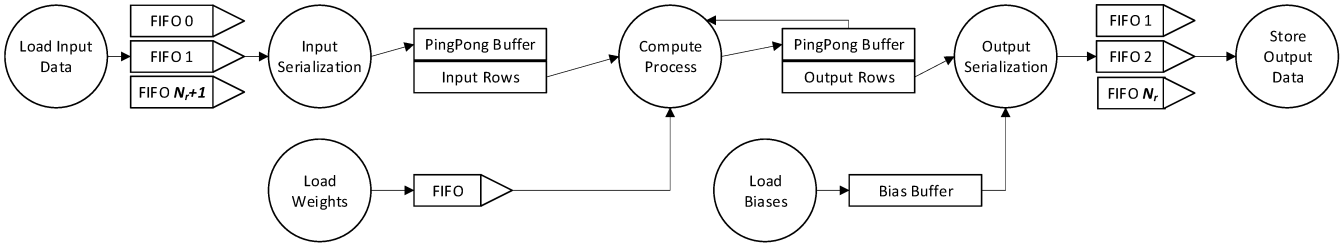

Fig. 3. Block diagram of the accelerator with main processes, buffers, and data flow.

multiplications, output transforms, and blocks of output row buffers by a factor of U_o. Second, multiple sets of input samples, denoted by U_i, can be used to compute their contributions to the same output features in parallel. This increases the number of multiplications and input transforms done by a factor of U_i. It also adds a step where the contributions of multiple input features are summed together before they are accumulated with the previous output sample. This step is done before the output transform, which is possible as it is a linear operation. In this case, it costs only 16 additions per 4×4 matrix of results instead of the 28+4 additions it would take for the added output transform and subsequent summation. Increasing U_i, however, does not cost more on-chip memory for input row buffers, as they are instantiated anyway to store a set of rows from all input features. To reduce logic utilization further, the summation can be done using the DSP's dedicated output carry path and post-adder. This option does not consume any extra logic resources for summing the intermediate contributions, but it restricts the FPGA router's freedom in placing the DSP slices.

D. Bandwidth Matching

Fig. 3 shows the block diagram of the accelerator, including the concurrent load and store processes together with the compute core and buffers. To continuously operate the compute core and match the memory's bandwidth, we combine different solutions. First, the input and output row buffers are instantiated twice as ping-pong buffers. While one buffer is being used by the compute process, a new set of data is written to the other buffer by the Input Serialization process. And when both processes are done, they switch buffers. This allows both processes to operate concurrently. Second, each row buffer has only a small number of data ports, giving it a much smaller bandwidth than what the HPM provides. To address this bandwidth mismatch, we use a solution similar to the coarse-grained approach presented by Cong et al. [16]. We write data in series and with high bandwidth into a set of N_r+2 512-bit wide, but shallow, FIFOs. Since these FIFOs do not have to be deep, they can be implemented with distributed RAM to spare BRAM resources. The input serialization process reads from all these FIFOs in parallel and writes the data serially into the row buffers. This results in a total bandwidth that is N_r+2 times the bandwidth of a single row buffer. If more bandwidth is needed, for example, when a large amount of unrolling is applied, this set of FIFOs and serialization process can be instantiated multiple times. Each serialization process can then write to a separate block of input buffers. We use a similar technique to write the output values with high bandwidth to the memory. In this case, we use a set of N_r shallow FIFOs. All FIFOs are filled in parallel with data from the output buffers. This data is then written into the memory with a bandwidth that matches the HPM, by emptying each FIFO one-by-one. The output serialization process also adds the output feature's bias value to each output sample and applies the Rectifier Linear Unit (ReLU) operation. Then, it writes the sample to the FIFO. This process is implemented for each row in parallel. The bias addition and ReLU operations are done in the output serialization process because all output samples pass through this process once, and only after all contributions have been added. Therefore, applying the bias here adds a minimal amount of logic and can be integrated into the process' pipeline, where it costs virtually no loss in throughput or latency.

E. Latency Hiding

HPM generally has a higher latency than standard DRAM. On Ultrascale FPGAs, DRAM can have an access latency as low as 40 cycles [17]. When using HBM, however, this latency is at least 110 cycles [11]. In our design, the latency measured to HMC is on the order of 100s of cycles, and depends significantly on packet size. To ensure that the processes that access memory do not suffer from this higher latency, it is important that they can continuously send memory instructions, without having to wait for any process flow-control to make them stall. For example, the *Load Input Data* process should not have to wait for the rest of the accelerator to be done processing a set of rows before starting to load the next set of rows and wait many cycles for data to arrive. To achieve this, all processes that load and store data are completely independent. The algorithm's execution is deterministic. By providing these processes with the feature size and number of features, they can continuously send memory instructions. Back-pressure is applied when the FIFOs are full or empty. The *Load Biases* process is an exception to this, as the bias data is only a small amount of data and is loaded in parallel with the first set of rows, after which the process is done.

F. Fully Connected Layers

We designed the accelerator with the ability to execute matrix-vector operations to compute the FC layers, which are

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

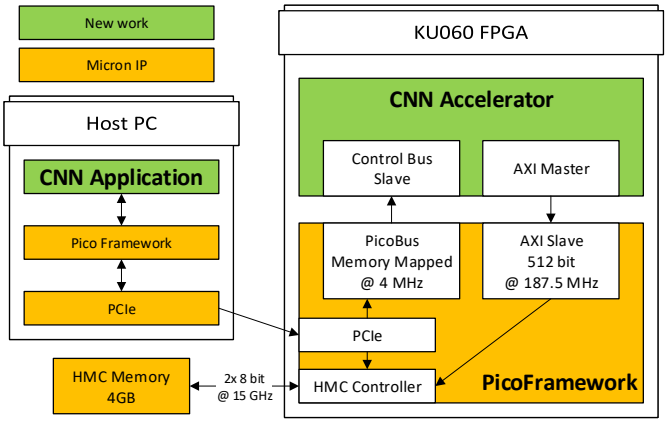

Fig. 4. System block diagram with host computer, FPGA, and data buses.

common in CNNs. This is done by bypassing the transforms and reusing the DSP blocks, as shown in Fig. 2. This adds minimal logic. We also designed it to achieve the maximum read bandwidth available because the FC operation is limited by the memory read bandwidth. During convolution, one set of weights is multiplied with many data samples. During a FC layer, however, the input samples are a vector that is multiplied with a large matrix of weights. Hence, one input sample is multiplied with many weights, which is the reverse of the data flow behavior during convolution. To efficiently execute the FC operations we reversed the role of weight and input data buffers. The large row buffers, and the high-bandwidth data-path to those buffers, are used for the FC weights, while the input vector is loaded using the convolution weight data-path. The large weight matrix is split in blocks and loaded into the row buffers, where each row holds part of a matrix column. The input and output transforms are bypassed, making the multiplication and summation steps in Fig. 3 operate essentially as a small matrix-vector multiplication core. A single block of output row buffers can store the entire output vector of all FC layers, so that it is only written to memory after the FC operation is completed. This allows us to further reduce the logic by ignoring the output from the multipliers that are added when $\boldsymbol{U_o}$ is increased. Since the FC operation is limited by the memory's read bandwidth, leaving those additional multipliers unused has no impact on performance.

IV. Demonstration System

A. PicoFramework

We implemented the proposed architecture's performance on a AC-510 [14] module from Micron. This module, which consists of a Xilinx Kintex Ultrascale 060 FPGA and a HMC unit, is connected to a host computer through a PCIe link, as shown in Fig. 4. The AC-510 module also includes Micron's PicoFramework, which consists of both an API and driver on the Host PC, as well as, an IP core on the FPGA. The PicoFramework handles all communications between the FPGA and host PC, and also between the accelerator and HMC memory. The HMC is a 4GB module and is connected to the FPGA by two uni-directional 8-bit serial links, operating at 15Gb/s per link. This provides a raw bandwidth of 15GB/s in each direction. The PicoFramework hides the HMC protocol and exposes a standard 512-bit AXI4 interface to the accelerator for access to the memory. This AXI port is clocked at 187.5 MHz which, therefore, limits the memory bandwidth in each direction to 12GB/s. The PicoFramework also exposes a memory mapped interface, called PicoBus by Micron, which allows for easy control of the accelerator from the host PC through a register interface. The PicoBus is clocked at only 4MHz and should not be used for data transfer. To transfer data between the host PC and FPGA, the PicoFramework provides a streaming interface which we use only to directly access the HMC from the host PC.

B. Accelerator Configuration

To maximize performance we configure the accelerator as follows. $\boldsymbol{N_r}$ is set to 14, as this is the greatest common denominator of VGG16's feature sizes and gives the maximum use of overlapping in the Winograd algorithm, while allowing each row buffer to be fully utilized in each layer. Both the input and output row buffers can store up to 1792 samples (equal to 8 of the largest features, or 128 of the smallest features). This value allows the accelerator to compute the contributions of a reasonable number of input features without having to switch to a different block of input rows. If the buffers are set to be too big, then after the last block of output data is computed, there will be a significant latency for that block to be written to memory. With a buffer capacity of 1792, the number of input buffer blocks $\boldsymbol{N_{bi}}$ must be set to 8 to be able to store a complete set of input features. For example, the input features to Layer 2 consist of 64 224×224 features, allowing $1792/224 = 8$ sets of rows to be stored per input buffer block and requiring $64/8 = 8$ input buffer blocks to store a set of rows from each feature.

V. Experimental Results

A. Execution time

To evaluate the performance of the accelerator when executing the VGG16 network, we measured the execution time and bandwidth per layer for various values of $\boldsymbol{U_i}$ and $\boldsymbol{U_o}$by counting the number of cycles and bytes transferred on the FPGA, respectively. The results are shown in Fig. 5(a-b). The spiking behavior in Layers 2-10 is caused by the fact that the output results are max-pooled after Layers 2,4,7, and 10. The max-pooling operation takes the maximum value from each 2×2 grid in the feature set and discards the other three values, dividing the size of the feature set by 4. The layers after the max-pool operation then double the number of features, resulting in half the amount of computation that need to performed in Layers 3, 5, and 8 compared to the previous layers. This is also reflected in the execution time, which is roughly halved in those layers compared to the pooled layers. Layers 11-13 are an exception because the number of features is not doubled and the amount of computation is one-fourth of that in Layer 10.

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

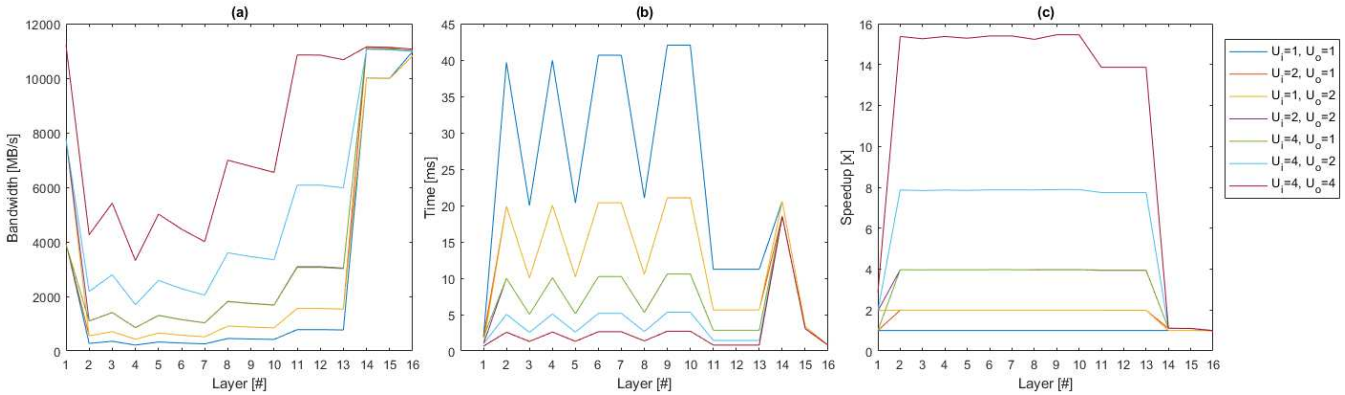

Fig. 5. Measured results for various values of U_i and U_o. (a) Average total bandwidth per layer. (b) Execution time per layer. (c) Speedup per layer referenced to U_i=U_o=1.

B. Near-Perfect Scaling

Fig. 5(a) shows that during the FC operation in Layers 14-16 the output bandwidth saturates at ∼11GB/s for all configurations with $U_i > 1$. This shows that the accelerator can fully saturate the read bandwidth during FC operations. Hence, higher performance requires more memory bandwidth. The bandwidth is not fully saturated when $U_i = 1$ because in this configuration the compute core processes exactly 512 bits of FC weights per cycle. Each time a block of FC weights is processed the input ping-pong buffer is switched. This requires the compute pipeline to be first emptied and then filled again, causing a slight performance penalty. Setting U_i to be greater than 1 increases the compute core's performance enough to completely hide this penalty, as the compute core can processes weights faster than they are loaded. When U_i=U_o= 4, the write bandwidth also saturates during Layer 1, and the read bandwidth saturates in Layers 11-13. The write bandwidth saturates because Layer 1 has only 3 input features, which results in a small amount of computation being done to obtain 64 output features. In Layers 11-13, the bandwidth is high due to the large number of weights being processed rapidly as the features are only 14×14 elements. The accelerator has to load 16 new sets of weights every 7 cycles, causing a peak bandwidth requirement of 13.7 GB/s.

Fig. 5(c) shows the relative speedup for each layer and configuration, compared to the configuration U_i=U_o= 1. Ideally the speedup is $U_i \times U_o$ during convolution layers. This is achieved in almost all cases, except for Layer 1 due to the small amount of computation done in that layer, and Layers 11-13 when U_i=U_o= 4. The latter is caused by the fact that the amount of data is so small that the compute process can complete the entire layer in a single iteration, thus preventing the accelerator from hiding memory latency through use of concurrent load, compute, and store processes [18]. Increasing U_i or U_o has virtually no effect because the FC layers are bandwidth limited, Fig. 6 shows the aggregate speedup of the convolutional layers against ideal scaling. The figure clearly shows that the accelerator performance increases almost perfectly. The non-ideal scaling in Layers 1 and 11-13 has only a small effect because most time is spent in Layers 2-10.

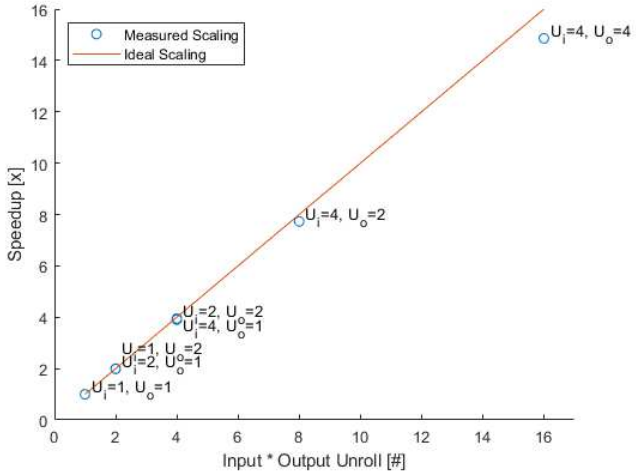

Fig. 6. Aggregate speedup of convolution layers for various unroll configurations. Measured by dividing the total execution time of layers 1-13 in configuration U_i=U_o= 1 by that of the other configurations.

Increasing the accelerator's performance by increasing the parallelism has diminishing returns. For the U_i=U_o= 4 configuration, the FC layers account for 48% of the total network execution time. Reducing the FC layer execution time requires more memory bandwidth. In cases where network layers have small feature sizes and many features, similar to Layers 11-13 of the VGG16 network, the required bandwidth to load the weights can also become a bottleneck. It is important to consider the memory access bandwidth when designing a highly parallel CNN accelerator, especially if the memory needs to be shared with other cores on the FPGA.

C. Hardware Cost

Fig. 7 shows that the hardware utilization does not change much across different configurations of U_i and U_o, except for the DSP block utilization. Between the configurations with U_i=U_o= 1 and U_i=U_o= 4 the LUT and FF utilization increases with only ∼10% of the available resources. Most of the BRAM is used for input row buffers and the HLS AXI

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

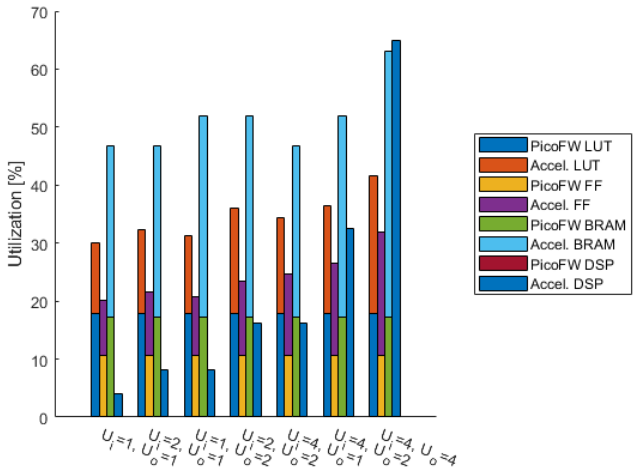

Fig. 7. Hardware utilization for various Accelerator Configurations. The PicoFramework and HMC controller utilization is the same for all configurations, and is included in the figure. Together they consume 17.78% of LUTs, 10.65% of FFs, and 17.28% of BRAMs.

ports. BRAM utilization increases only when increasing $\boldsymbol{U_o}$ to store the additional blocks of output features, which consume 4.4%$\times\boldsymbol{U_o}$. Unfortunately, increasing $\boldsymbol{U_i}$ or $\boldsymbol{U_o}$ above 4 was not feasible because the FPGA routing becomes congested, mainly due to the process control logic generated by Vivado HLS. An RTL implementation of this design is expected to give better results and allow for use of the entire FPGA.

D. Comparison to Prior Works

Table I contrasts our work with prior works. While using all available DSPs in the FPGA, the accelerator demonstrated by Ma et al. [6] does not reach the performance of our accelerator during convolution. Its total latency is a bit lower due to the higher memory read bandwidth the FPGA can achieve. Our accelerator, however, needs only 59% of the DSP resources, thanks to the use of the Winograd Algorithm. Our accelerator performs similarly to the accelerator demonstrated by Huang et al. [8]. The throughput per DSP in the VU440 FPGA is significantly higher than our throughput per DSP, due to the higher read bandwidth it has available during the FC layers.

TABLE I
COMPARISON WITH PRIOR WORKS

Reference	[6]	[8]		**This work**
FPGA	Arria 10	Xilinx	Xilinx	Xilinx
FPGA	GX1150	VX690T	VU440	KCU060
Network	VGG-16			
CNN Algorithm	Conventional	Winograd F(3×3,2×2)		
Frequency [MHz]	200	150	200	187.5
Max. Memory Bandwidth [GB/s]	20 Read+Write	12.8 Read+Write	20 Read+Write	12 Read 12 Write
DSP Utilization	3036(100%)	1402(39%)	1402(49%)	1792(64.9%)
Latency [ms]	42.98	62.65[a]	40.99[a]	44.92
CNV Latency [ms]	26[b]	-		23.1
Throughput [GOP/s]	720	494	755	689
Throughput/DSP [GOP/s/DSP]	720 0.24	494 0.35	755 0.53	689 0.38

[a] Calculated from reported throughput
[b] Number read from Fig. 10 in the reference

VI. CONCLUSIONS

We presented the first scalable architecture for CNN accelerators that leverages stacked-DRAM HPM. The accelerator performance scales near-perfectly with increased use of parallelism. The saturation of the memory link shows that the accelerator makes full use of the HPM's bandwidth, while effectively hiding its access latency. Furthermore, we have shown that increasing the parallelism of CNN accelerators has diminishing returns, and matrix-vector multipliers has virtually no returns, if the available memory bandwidth is not increased significantly with the help of emerging HPM technologies.

Acknowledgments. This research was supported in part by the National Science Foundation (A#: 1764000) and the Department of Energy (DOE) Small Business Innovation Research (SBIR) ASCR Program under contract DE-SC0017182.

REFERENCES

[1] K. Abdelouahab et al. Accelerating CNN inference on FPGAs: A Survey. *arXiv preprint arXiv:1806.01683*, 2018.
[2] K. Guo et al. A Survey of FPGA Based Neural Network Accelerator. *arXiv preprint arXiv:1712.08934*, 2017.
[3] S. Venieris, A. Kouris, and C.-S. Bouganis. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Computing Surveys (CSUR)*, 51(3):56, 2018.
[4] D. Giri et al. ESP4ML: platform-based design of systems-on-chip for embedded machine learning. In *Proc. of the Conf. on Design, Automation, and Test in Europe (DATE)*, March 2020.
[5] C. Zhang et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proc. Intl. Symp. on Field-Programmable Gate Arrays*, pages 161–170, 2015.
[6] Y. Ma et al. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Intl. Conf. on Field Programmable Logic and Applications*, pages 1–8, 2017.
[7] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *Proc. of the Conf. on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.
[8] Y. Huang et al. A high-efficiency FPGA-based accelerator for convolutional neural networks using Winograd algorithm. In *Journal of Physics: Conference Series*, volume 1026, page 012019. IOP Publishing, 2018.
[9] Xilinx Inc. Memory Solutions - External Memory Interfaces. https://www.xilinx.com/products/technology/memory.html\#externalMemory.
[10] Inc. Micron Technology. Hybrid Memory Cube – HMC Gen2. https://www.micron.com/-/media/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf.
[11] Xilinx Inc. AXI High Bandwidth Memory Controller v1.0. https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.
[12] S. Winograd. *Arithmetic complexity of computations*. Siam, 1980.
[13] L. P. Carloni. From latency-insensitive design to communication-based system-level design. *Proc. of the IEEE*, 103(11):2133–2151, November 2015.
[14] Micron Technology Inc. AC-510. https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/ac-series-hpc-modules/ac-510.
[15] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[16] J. Cong et al. Bandwidth optimization through on-chip memory restructuring for HLS. In *Design Automation Conference (DAC)*, pages 1–6, 2017.
[17] Xilinx Inc. UltraScale Architecture-Based FPGAs Memory IP v1.4. https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf. Accessed: 2019-11-26.
[18] P. Mantovani, G. Di Guglielmo, and L. P. Carloni. High-level synthesis of accelerators in embedded scalable platforms. In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 204–211, January 2016.

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