Optimized FPGA-based Deep Learning Accelerator for Sparse CNN using High Bandwidth Memory

Chao Jiang
*SHREC**
University of Florida
Gainesville, Florida, US
jc19chaoj@ufl.edu

David Ojika
*SHREC**
University of Florida
Gainesville, Florida, US
davido@ufl.edu

Bhavesh Patel
Dell EMC
Dell Technologies
Texas, US
bhavesh.a.patel@dell.com

Herman Lam
*SHREC**
University of Florida
Gainesville, Florida, US
hlam@ufl.edu

***Abstract*—Large Convolutional Neural Networks (CNNs) are often pruned and compressed to reduce the amount of parameters and memory requirement. However, the resulting irregularity in the sparse data makes it difficult for FPGA accelerators that contains systolic arrays of Multiply-and-Accumulate (MAC) units, such as Intel's FPGA-based Deep Learning Accelerator (DLA), to achieve their maximum potential. Moreover, FPGAs with low-bandwidth off-chip memory could not satisfy the memory bandwidth requirement for sparse matrix computation. In this paper, we present 1) a sparse matrix packing technique that condenses sparse inputs and filters before feeding them into the systolic array of MAC units in the Intel DLA, and 2) a customization of the Intel DLA which allows the FPGA to efficiently utilize a high bandwidth memory (HBM2) integrated in the same package. For end-to-end inference with randomly pruned ResNet-50/MobileNet CNN models, our experiments demonstrate 2.7x/3x performance improvement compared to an FPGA with DDR4, 2.2x/2.1x speedup against a server-class Intel SkyLake CPU, and comparable performance with 1.7x/2x power efficiency gain as compared to an NVidia V100 GPU.**

***Index Terms*—Sparse CNN, FPGA, Deep learning accelerator, High bandwidth memory**

I. Introduction

Machine learning using Convolutional Neural Networks (CNNs) has emerged as powerful tools for almost every domain involving analytics of big data. Advances in CNNs, in the form of Deep Neural Networks (DNNs), have resulted in highly accurate predictions in image classification and object detection applications [1], [2]. However, the great success in CNN comes with the cost of excessive computation and memory bandwidth requirement. The state-of-the-art CNN models, e.g. VGGNet [3], GoogLeNet [4], MobileNet [5] and ResNet [6], comprise of hundreds of layers, each of which contains numerous filters with hundreds of millions of weights, costing billions of arithmetic operations for analyzing and extracting features from input images.

With recent progress in FPGA technology, FPGAs have been gaining popularity as hardware accelerators to improve the computation efficiency of CNN models. FPGAs have many advantages over traditional CPU architectures and other accelerators such as deterministic low latency, energy efficiency, and reconfigurability. The performance of recent FPGAs is reportedly to be comparable to GPU performance recently with 9.2 TFLOPS for Intel Stratix 10 FPGA [7] and up to 40 TFLOPS for Intel Agilex FPGA [8]. Moveover, efficiency of data transfering between FPGA and external memory, which is often the biggest shortcoming of FPGA-based accelerators, has also been improving with High Bandwidth Memory (HBM2) integrated with the FPGA die in the same package, such as Intel Stratix 10 MX FPGA and Xilinx Virtex UltraScale+ with HBM2 are now available. As a result, there have been much interest in FPGA-based CNN accelerators, both in academia and industry. Successful examples include PipeCNN [9], hls4ml [10] from Fermilab, Xilinx's Deep Neural Network Development Kit (DNNDK) [11], and Intel's OpenVINO toolkit [12].

On the algorithm side, various model pruning and compression methods have been introduced to reduce the amount of weights by leveraging the intrinsic redundancy in the weights of CNN models. These methods can result in very high sparsity in the convolutional filter tensors without sacrificing much model inference accuracy [13]–[16]. By doing so, it reduces the memory requirement and arithmetic operations needed for inference, as sparse matrices can be compressed and operations with zeros can be skipped.

However, high sparsity does not necessarily guarantee that the inference of a sparse CNN would achieve better performance than its dense counterpart. Due to the irregularity of the sparsity, it is often difficult for hardware accelerators to achieve optimal performance and efficiency, especially ones with systolic array of computation units which are widely adopted in many FPGA-based Deep Learning Accelerators cited above. Systolic array architectures are usually efficient for DNN applications because the local data shifting naturally mimics the inherent data movement of a 2D convolution. Systolic arrays also leverage the abundant data reuse in DNNs while keeping the processing elements busy to produce high throughput. However, because zeros in pruned filter matrix can be distributed in an unstructured manner, making it challenging to efficiently utilize the regular structure of a systolic array. When inferencing sparse CNN models, zeros in the filter matrix still occupy the computation units in the systolic array, resulting in sub-optimal efficiency.

To solve this problem, in this work, we present an opti-

* SHREC: NSF Center for Space, High-Performance, and Resilient Computing

mization for the systolic array architecture of deep learning accelerators for sparse CNN models on FPGA platforms. Further enhancement results from the use of an integrated high bandwidth memory (HBM2) for efficient off-chip communication which accelerates memory-bound operations for deep learning accelerators. The main contributions of this work are:

- We propose a sparse matrix packing method that condenses sparse filters to reduce the computation requirement for systolic array accelerators. A bitmap representation is used to indicate positions of zero and non-zero elements in the sparse filter. The bitmap representation enables efficient extraction of feature map inputs matching the position of non-zeros in the sparse filter.
- We analyze the bandwidth requirement of our optimization for sparse CNNs and present efficient implementations of the memory accessing module to utilize HBM2 bandwidth in OpenCL.
- Under NDA, we obtained the source code and customized the Intel Deep Learning Accelerator (DLA) to integrate the proposed sparse matrix packing method into the DLA architecture on the FPGA;
- Our custom DLA was ported to a platform using the Intel Stratix 10 MX FPGA with HBM2. Evaluation was performed for sparse MobileNet and ResNet-50 CNN models, resulting in 2.06x/3.44x performance gain in DLA's systolic array computation module, 2.2x/2.1x speedup against a server-class Intel SkyLake CPU, and comparable performance with 1.7x/2x energy efficiency gain as compared to an NVidia V100 GPU.

The remainder of the paper is organized as follows. Section II introduces related research in sparse matrix accelerators and FPGA-based deep learning accelerators. Section III presents the proposed optimization for systolic array based deep learning accelerators and the use of HBM2 memory to accelerate memory-bound operations. Section IV and Section V describes experimental setup and results, providing comparison of FPGAs (with DDR4 and HBM2 memory) against CPU and GPU for inferencing sparse CNN. The paper is concluded in Section VI.

II. Related Works

A. Sparse Matrix Accelerators

One of the main approaches to improve DNN performance is to exploit the high degree of redundancy in the weights parameters of DNN models [13]–[17]. Pruning techniques, such as Deep Compression [13] and Dynamic Network Surgery [18], can efficiently compress a DNN model to only a fraction of its original size. These pruning techniques have led to an increased interest in sparse matrix algorithms, as the dense weight matrices are pruned to sparse matrices. Due to irregular patterns that are inherent in sparse data structures, conventional architectures, such as CPU and GPU, struggle to achieve a similar level of performance on sparse matrices comparing to their performance on dense matrices. Thus, there is growing interest in accelerating sparse matrix through custom hardware accelerators.

Early implementation of sparse matrix accelerators, such as the DAP processor [19], focused on mainly on sparse matrix multiplications for scientific and engineering applications. OuterSPACE [20] is a co-designed solution that accelerates sparse matrix multiplication by reconfiguring on-chip memory. ExTensor [21] uses hierarchical intersection detection to accelerate sparse matrix algebra. However, neither OuterSPACE and ExTensor was designed for DNN applications. Efficient Inference Engine [22] is a hardware accelerator specific to DNNs. But it relies on complex central scheduling units to coordinate the computation units. SparTen [23] accelerates CNNs by designing asynchronous computation units for exploiting sparsity in both feature maps and filters.

Instead of redesigning computation units specially just for sparse matrix, our work supports efficient sparse (in additional to dense) matrix vector multiplication on a systolic array with minor modifications to support end-to-end inference of sparse CNN on FPGA-based deep learning accelerators.

B. FPGA-based Deep Learning Accelerators

Although CPUs and GPUs have been widely used for DNN inferencing, inference engines accelerated with FPGAs have recently emerged. Recent improvements in FPGA technologies greatly increased the performance for DNN applications. Furthermore, FPGAs have other advantages important to many mission-critical applications such as low latency and energy efficiency. As a result, the amount of research and development on deploying and accelerating DNN models on FPGAs in recent years has grown, demonstrating great interest in both academia and industry. While some of the works focused on optimizing datapaths or computation algorithms for FPGA devices, many also involve developing tools for DNN model inferencing on FPGA platforms to provide a generalized framework for developers to build their customized applications.

1) PipeCNN [9]: One notable tool developed in the research community is PipeCNN, an OpenCL-based FPGA accelerator designed for large-scale convolutional neural networks (CNNs). The main goal of PipeCNN is to provide an FPGA accelerator architecture of deeply pipelined CNN kernels to achieve improved throughput in the inference stage. Unlike previous OpenCL design, memory bandwidth is minimized by pipelining CNN kernels. Efficiency is enhanced by using task-mapping techniques and data reuse. The PipeCNN architecture was verified by implementing two CNNs, AlexNet and VGG, on an Altera Stratix-V A7 FPGA, achieving a peak performance of 33.9 GOPS with a 34 percent resource reduction on DSP blocks.

2) hls4ml [10]: Another notable FPGA-based inference tool is hls4ml from Fermilab, which is a deep neural network compiler based on HLS (High-level Synthesis language). The input to hls4ml is a fully connected neural network trained from conventional training frameworks such as Keras and PyTorch. The network is translated to Vivado HLS (from Xilinx) and then compiled for the target FPGA. For the initial result in using this framework, the researchers focused on

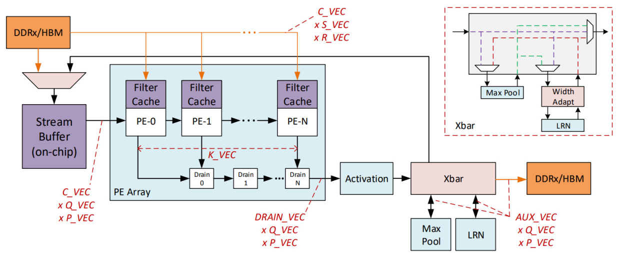


Fig. 1. Intel Deep Learning Accelerator architecture.

using FPGAs for machine learning in an application of real-time event reconstruction and filtering in the Large Hadron Collider at CERN. The accessibility and ease of configurability in HLS allows for physicists to quickly develop and optimize machine learning algorithms targeting FPGA hardware.

3) Xilinx DNNDK [11]: With the recent acquisition of DeePhi, Xilinx provides the Deep Neural Network Development Kit (DNNDK) to enable the acceleration of the deep learning algorithms in FPGAs and SoCs. At the heart of the DNNDK is the deep learning processor unit (DPU). The DNNDK deep learning SDK is designed as an integrated framework which aims to simplify and accelerate deep learning applications development and deployment for Xilinx DPU platforms.The basic stages of deploying a deep learning application into a DPU are: compress the DNN model to reduce the model size without loss of accuracy; compile the DNN model into DPU instruction code; create an application using DNNDK (C/C++) APIs; use the hybrid compiler to compile and deploy the hybrid DPU application on the target DPU platform.

4) Intel OpenVINO and Intel DLA [12], [24]: Intel OpenVINO is a comprehensive toolkit designed to support deep learning, computer vision, and hardware acceleration using heterogeneous (CPU, GPU, FPGA) platforms. The OpenVINO software is built to emulate the Open Visual inference and neural network optimization. It extends the workload across Intel hardware and maximizes performance. The OpenVINO toolkit comprises of a Model Optimizer and an Inference Engine.

The Model Optimizer is a cross-platform, command-line tool that facilitates the transition between the training and deployment environment on a target inference engine. The Model Optimizer takes, as input, a trained deep-learning model outputted from one of the supported frameworks (e.g., TensorFlow, Keras). It performs static model analysis and adjusts the deep learning model for optimal execution on end-point target devices, CPU, GPU, FPGA, or HETERO (CPU+GPU or CPU+FPGA). The output of the Model Optimizer is an Intermediate Representation (IR) suitable as input to the selected target Inference Engine. The Inference Engine is a C++ library with a set of C++ classes to infer data (images) to obtain a result. The C++ library provides an API to read the IR, set the input and output formats, and execute the model on devices. In our study, our goal in the Deployment and Inferencing stage

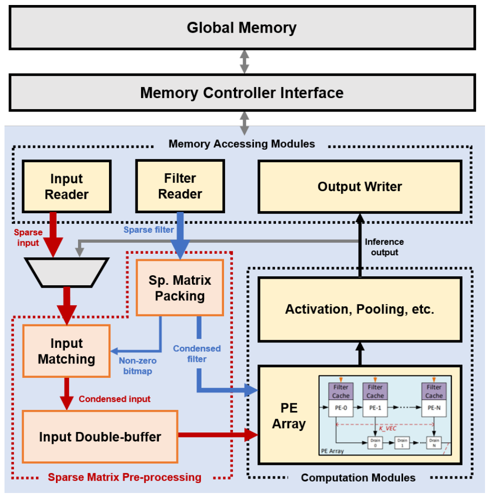


Fig. 2. Overall architecture of proposed optimization of sparse CNN accelerator.

is to deploy the trained model on an FPGA to accelerate the classification process.

The Intel Deep Learning Accelerator developer suite or Intel DLA is the underlying tool that enables the inferencing of DNN models on FPGAs with OpenVINO. DLA consists of an API layer called DLIA plugin that interacts with OpenVINO's inference engine and an FPGA bitstream that creates the accelerator architecture shown in Fig. 1. Intel DLA can perform end-to-end DNN inference and is designed to be adapted for various models. Our work leverages Intel DLA as our base design, which is then customized to accelerate sparse CNNs on Intel FPGA platforms.

III. Customized Intel Deep Learning Accelerator for Sparse CNN

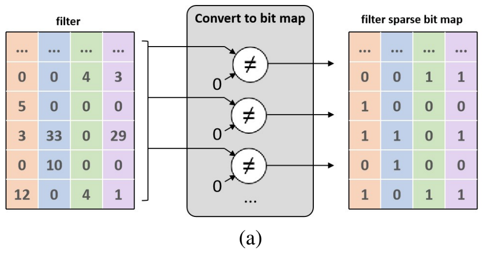
In this section, we describe the proposed pre-processing method of packing sparse matrix to optimize sparse CNN inference for the systolic array computation module of Intel Deep Learning Accelerator.

A. Overall Architecture and Sparse Matrix Packing

Fig. 2 shows the top-level diagram of the custom Intel DLA architecture for accelerating inference of sparse CNN models on systolic array accelerators. The architecture is divided into three main module blocks: 1) *Memory Accessing Modules*, 2) *Sparse Matrix Pre-processing Modules*, and 3) *Computation Modules*.

The *Memory Accessing Modules* interact with global off-chip memory through an Input Reader and a Filter Reader reading feature map inputs and filters for convolutional operations. It also contains an Output Writer writing the final inference output back to the memory.

The *Computation Modules* block consists of a systolic PE Array which utilizes FPGA's DSP resources to perform floating-point Multiply-and-Accumulate operations for convolutional layers. The size of the PE Array can be configured to fit different FPGAs and CNN models. The output of the



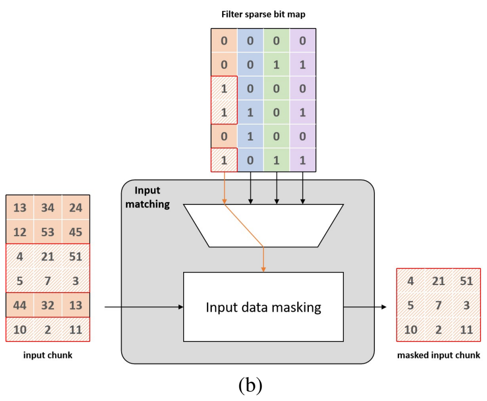


Fig. 3. (a) Representing positions of zero and non-zero numbers with sparse bitmap. (b) Compress input feature maps based on the filter sparse bitmap.

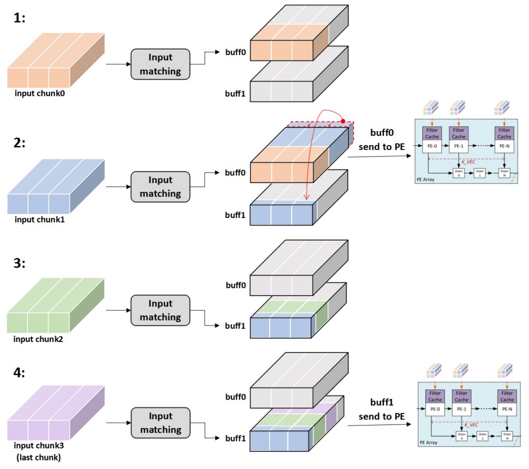


Fig. 4. Use double buffering for condensed input data.

PE Array goes through a series of secondary computation modules with ReLU activation, pooling, normalization, and custom compute primitives for DNN models.

The *Sparse Matrix Pre-processing Modules* block consists of a Sparse Matrix Packing module for compressing sparse filter into a condensed format by eliminating zero elements and creating a sparse bitmap for the Input Matching kernel to extract corresponding input data which will be stored into an on-chip buffer before streaming to the PE Array. Different from traditional sparse tensor representations, such as Compressed Sparse Row (CSR) or Coordinate List (COO) [25]–[29] which represent indices of non-zero values as integer numbers, we use a sparse bitmap representation as introduced in paper [30], with binary number ”0”s and ”1”s to represent positions of zero and non-zero numbers respectively. The benefits of sparse bitmap representation are 1) it is more memory efficient than integer indices and 2) binary operations can be easily implemented on FPGA devices. Fig. 3a demonstrates the sparse bitmap conversion operations and Fig. 3b demonstrates the index matching process where the Input Matching kernel masks the original input chunk with the filter sparse bitmap, extracting the input data that matches the position of 1’s.

B. Sparse Input Double-buffering

Due to the unpredictability of the sparsity for a given sparse matrix, using single buffer may result in the buffer being unfilled or overflowed which can cause under-utilizing the PE computation module or losing the overflowed data. Thus, we employ double buffering in the custom DLA by implementing two identical buffers, where one buffer sends data to the PE Array only when it is filled while the other holds overflowed data.

Fig. 4 demonstrates the usage of double buffering. As described in above section, an sparse input vector is firstly passed into the Input Matching kernel to extract the data that matches the position of the filter’s sparse bitmap. The resu lting condensed input will be stored in ”buff_0” as shown in the step 1. If ”buff_0” is not filled, the next input chunk will be processed in the same manner and added to ”buff_0”. Once it is filled, the data in ”buff_0” will be sent to the PE Array to perform convolution operations. If ”buff_0” cannot hold all input data received in the current iteration, the second buffer ”buff_1”, will be used to hold the overflowed data. Once the PE has received the data, the next input chunk will be buffered in ”buff_1” instead and ”buff_0” will be reset to store overflowed data. After all the input chunks have been processed, a ”last_input” flag will be asserted, which allows the remaining data in either buffers to be sent to the PE. Note that in this scenario, the PE Array only performed convolution operation twice with four input chunks received from the Input Reader, thus reducing the computation requirement by a half.

C. Bandwidth Analysis and Integration of HBM2 Memory

Although our sparse matrix packing optimization condenses sparse matrix and reduces the computation requirement, the custom DLA requires higher memory bandwidth when inferencing CNN models with high sparsity because the PE Array computation modules can be underutilized due to waiting for the buffer being filled from the memory-bound pre-processing stage.

We can estimate the minimum bandwidth requirement based on the sparsity of the filter data and the computation capacity of the PE Array. Assuming that sparsity of a CNN model is SP_{filter}, each PE is capable of computing dot products of N-bit floating-point vectors with a size of S_{dot} running at a clock frequency of f_{kernel}, the minimum memory bandwidth required BW_{min} for avoiding stalls in the PE Array is calculated in Eq. 1.

$$BW_{min} = S_{dot} \times \frac{1}{1 - SP_{filter}} \times N \times f_{kernel} \quad (1)$$

Assuming SP_{filter} = 70%, N = 16 (FP16 precision), S_{dot} = 16, f_{kernel} = 300 MHz, it requires a data bandwidth of at least 32 GB/s for both Input Reader and Filter Reader to prevent stalls in the PE Array. For SP_{filter} = 90%, the required bandwidth increases to 96 GB/s. Note that a DDR4 memory can only provide a maximum of 16.9GB/s bandwidth.

As mentioned in the previous section, recent FPGAs, such as Intel Stratix 10 MX, are equipped with integrated high bandwidth HBM2 memory. As shown in Fig. 5a, HBM2 uses 3D stacked DRAM dies connected with Through-Silicon Vias. Each DRAM die consists of two physical channels (CH 0 to 7), each of which can be further divided into two pseudo channels (PC 0 to 15). The HBM2 is integrated with the Intel Stratix 10 MX FPGA with Embedded Multi-die Interconnect Bridge (EMIB) in the same package. All physical channels and corresponding pseudo channels are connected through Intel HBM2 interface IP with dedicated memory controllers for each individual physical channels of the HBM2. By using the FPGA with HBM2, we can allow both Input Reader and Filter Reader kernels to read more data per cycle, thus further optimizing the performance of our custom DLA.

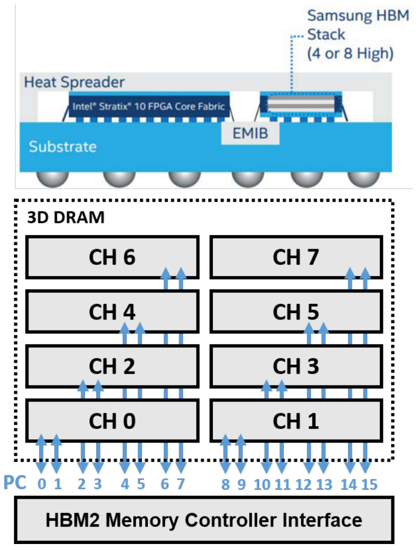
However, utilizing HBM2 memory in OpenCL is different from using traditional DDR memory where multiple memory banks are combined to form a single global memory system. For the HBM2 memory, OpenCL kernels interact with each HBM2 pseudo channel as individual memory system with independent address space. Thus, it requires a developer to explicitly specify which pseudo channel each global memory pointer should access in the OpenCL kernel code.

Similarly, in the OpenCL host code, it also requires that all buffer transfers must target the pseudo channels specified by the OpenCL kernel. Failure to do so can result in the OpenCL runtime allocating buffer in a wrong memory system and later having to copy the data to the correct memory system, which may greatly reduce the overall performance.

If the custom DLA is implemented on the FPGA+HBM2 device without being explicitly specified to use multiple HBM2 pseudo channels, the Memory Accessing Modules of the DLA will only be able to read data from the first pseudo channel of the first HBM2 physical channel (”PC 0” of ”CH 0” Fig. 5a) by default. Since a single HBM2 pseudo channel has 64-bit wide data I/O and with 1 GHz of maximum memory clock frequency [31], it can only provide up to 64 bits $\times$ 2 (double data rate) $\times$ 1 GHz = 16 GB/s memory bandwidth which will not satisfy the minimum memory bandwidth required for the custom DLA.

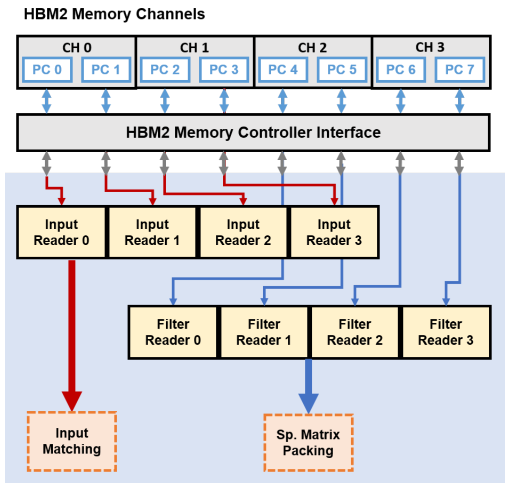
To use multiple HBM2 pseudo channels, we need to first partition the input data and filter data equally in the OpenCL host program for the number of pseudo channels that we decide to use for the Input Reader and the Filter Reader kernels and transfer the data buffers to their corresponding HBM2 pseudo channels.

For the OpenCL kernel to access multiple HBM2 pseudo channels, one of the approaches is to pass the partitioned input and filter data using multiple global memory pointers to the kernel functions of both Input Reader and Filter Reader, and assign each pointers to their corresponding pseudo channels. However, this method may greatly increase the complexity of the OpenCL kernel as accessing each additional HBM2 pseudo channel requires an additional memory accessing system to be instantiated by the OpenCL compiler, which makes it difficult to optimize the performance of the kernel. Thus, we proposed an alternative approach, as shown in Fig. 5b, that duplicates the Input Reader and the Filter Reader kernels to read partitioned data from different HBM2 pseudo channels for each replications. This approach not only keeps each kernel simple, but also allowing the partitioned input and filter data to be gathered and transferred through more efficient inter-kernel FIFO channels to the subsequent Sparse Matrix Pre-processing Modules.



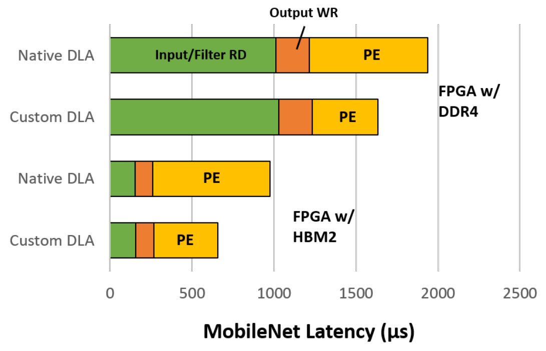
Samsung HBM Stack (4 or 8 High)
Heat Spreader
Intel® Stratix® 10 FPGA Core Fabric
EMIB
Substrate
3D DRAM
CH 6
CH 7
CH 4
CH 5
CH 2
CH 3
CH 0
CH 1
PC 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
HBM2 Memory Controller Interface

(a)

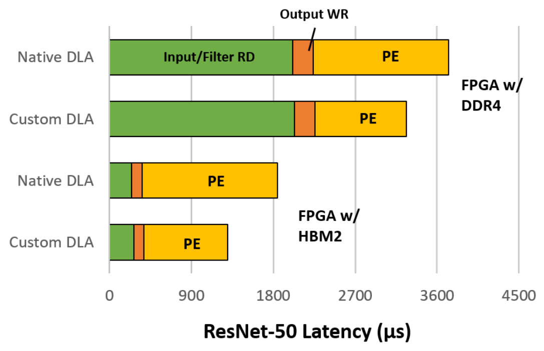


(b)

Fig. 5. (a) Intel Stratix 10 MX with HBM2 has 8 physical channels (CH) and 16 pseudo channels (PC). (b) Duplicate memory accessing modules to use multiple HBM2 pseudo channels.



(a)



(b)

Fig. 6. PE computation latency for FPGA+DDR4 vs. FPGA+HBM2.

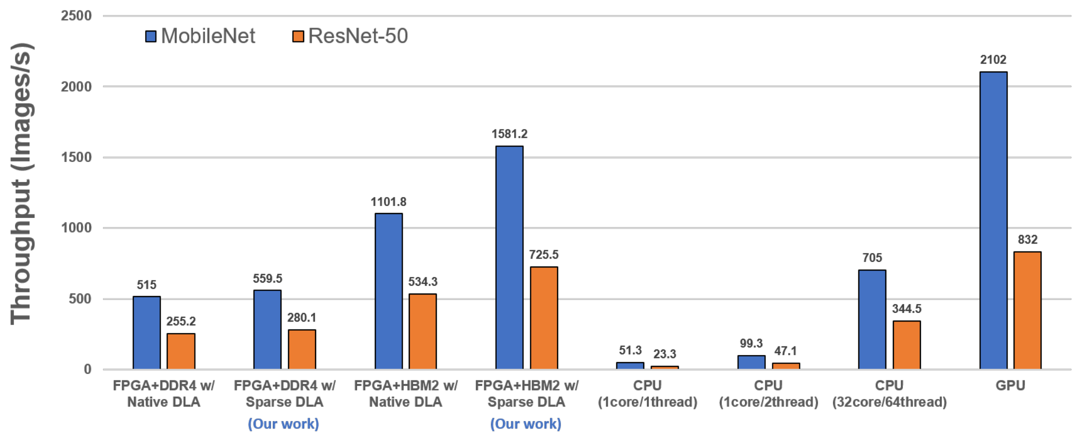


Fig. 7. Comparison of model inference throughput for FPGA, CPU and GPU.

TABLE I
SPARSE CNN BENCHMARK MODELS

Model	Sparsity	Parameters	GFLOP	Top-1 (Pruned)
MobileNet	59.3%	3.49M	0.62	70.85%
ResNet-50	60.4%	25.53M	6.99	74.56%

TABLE II
RESOURCE UTILIZATION OF SYNTHESIZED DLA

FPGA Platform	ALM	DSP	Freq.
Arria 10 w/ DDR4	279K (63%)	1044 (69%)	260 MHz
S-10 MX w/ HBM2	334K (45%)	1442 (36%)	257 MHz

IV. EXPERIMENTAL SETUP

In this section, we describe the experimental setup for the FPGA-based inferencing used in this study. The targeted FPGA experimental platforms are the Intel Programmable Acceleration Card (PAC) and BittWare 520N-MX acceleration board. The Intel PAC contains an Arria 10 GX1150 FPGA, a moderate-sized FPGA with 428K Adaptive Logic Modules (ALMs), 1518 DSPs and 8 GB DDR4 with a maximum bandwidth of 16.9 GB/s. The BittWare 520N-MX contains an Intel Stratix 10 MX2100 FPGA with 702K ALMs, 3960 DSPs and 16 GB integrated HBM2 memory with up to 410 GB/s bandwidth [31].

For fair comparison of the two FPGAs and study the performance difference with DDR4 and HBM2, we configure the custom DLA architecture to be identical for both FPGA boards. The custom DLA is configured to have a 2D systolic array of $16 \times 16 \times 64$ PEs that perform half-precision floating point (FP16) Multiply-and-Accumulate to compute convolution of 16×16 pixels in 64 output feature maps. For the FPGA+HBM2 device, we configure the Input Reader to read data from four HBM2 pseudo channels (PC 0 to PC 3) as well as the Filter Reader to read from another four (PC 4 to PC 7). Using Intel Quartus 19.4 with Intel OpenCL SDK for FPGA, the custom DLA is synthesized, placed/routed and the bitstream is uploaded to both FPGA boards.

Two CNN benchmark models are used: MobileNet and ResNet-50. The pruned versions of both models are obtained from the Intel OpenVINO Model Zoo [32]. Table I shows detailed specification for both models. The average sparsity of the pruned MobileNet and ResNet-50 are 59.3 % and 60.4 %, respectively [32]. Comparing to their original models tested on the ImageNet benchmarking dataset, the pruned models show 3.9 % and 2.6 % degradation in Top-1 accuracy, respectively [5], [6]. For comparing FPGAs against the CPU and GPU, we use a dual-socket Intel Gold 6130 SkyLake CPU (32 cores with 2 threads per core) and an NVidia Tesla V100 GPU with a 16 GB HBM2 memory providing up to 900 GB/s bandwidth [33]. We use Intel OpenVINO Toolkit for optimizing inferencing on both FPGA and CPU while using NVidia's TensorRT [34] for the GPU. Both CPU and GPU use FP32 precision during inferencing. For power comparison, we used BittWare Board Management Controller's power monitoring tool for the FPGA boards and used CUDA nvidia-smi utility for the Tesla V100 GPU.

V. Results and Analysis

In this section, we compare the performance results for the CPU, GPU, FPGA+HBM2 and FPGA+DDR4, with or without optimized DLA.

Table II shows resource utilization of the custom DLA on both FPGA+DDR4 and FPGA+HBM2. The custom DLA optimized for FPGA+HBM2 takes more resource due to the duplicated Input Reader and Filter Reader kernels (four Input Readers and four Filter Readers) with additional control logic and Load/Store Units (LSUs) being synthesized by the OpenCL compiler.

Fig. 6 shows the inference latency for both FPGA devices to study the bottleneck of the performance and the benefit of using the HBM2. We breakdown FPGA's inference latency into three stages: input reading stage (with sparse matrix pre-processing), PE computing stage, and output writing stage. As shown in Fig. 6a and Fig. 6b, memory-bound operations consume 75%/69% of total latency for the FPGA+DDR4 implementation for MobileNet and ResNet-50 respectively. Even with ∼2x improvement in PE computation latency using the custom DLA, the low bandwidth of DDR4 memory limits its overall inference performance. For the FPGA+HBM2 device, using the proposed kernel-duplication method for accessing HBM2 pseudo channels solves the critical bandwidth bottleneck, which significantly reduces the latency consumed in input reading and output writing stages. With only eight HBM2 pseudo channels, the custom DLA reduces the latency of memory-bound operations by 4.6x/6.1x, resulting in 2.7x/3x performance gain in overall inference latency than FPGA+DDR4 for MobileNet and ResNet-50.

Fig. 7 compares the FPGA performance with an Intel Xeon Gold 6130 SkyLake CPU and an NVidia Tesla V100 GPU. The FPGA with HBM2 memory outperformed CPU by 30.8x/29.8x, 15.2x/13.8x and 2.2x/2.1x for 1 core/1 thread, 1 core/2 threads and all 32 cores/64 threads, respectively, for MobileNet and ResNet-50. The Tesla V100 GPU outperformed the FPGA by 1.2x/1.3x but at a cost of higher power consumption. The FPGA achieved lower power consumption due to low resource utilization (45%) and with a clock frequency at only 257 MHz. Fig. 8 shows our proposed custom DLA using the FPGA+HBM2 device achieved 3.5x/3.3x more power efficiency than CPU with 32 cores/64 threads and 1.7x/2x more power efficiency than V100 GPU for MobileNet and ResNet-50, respectively.

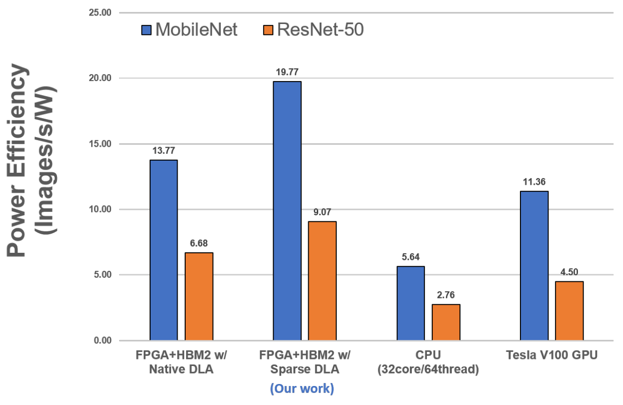


Fig. 8. Comparison of power efficiency for FPGA, CPU and GPU.

VI. Conclusion

FPGAs have been showing great potentials in recent years for accelerating CNN applications. However, accelerating sparse CNN models can be challenging due to the random distribution of zeros in the filters of the convolutional layers, which can result in inefficient utilization of computation resources in a typical systolic array accelerator. We propose a solution to this problem by carefully pre-processing both sparse input and filter tensors before sending them to the systolic array. By doing so, we can greatly reduce the effect of irregularity of a sparse CNN. Using the Intel Stratix 10 MX FPGA with HBM2, we are able to achieve up to 3x improvement compared to an FPGA with DDR4, producing speedup of 2.2x against 32 cores server-class Intel SkyLake CPU and comparable performance with 2x power efficiency against an NVidia Tesla V100 GPU.Note again that the two CNN benchmark models used, MobileNet and ResNet-50, have sparsity of 59.3 % and 60.4 %, respectively. CNN models with higher sparsity (e.g., sparsity greater than 90 % such as NLP models and Graph NNs) will benefit even more with our sparse matrix packing optimization. Since more zeros will be eliminated in the pre-processing stage, the corresponding waste of computation would be avoided.

Going forward, our sparse matrix packing pre-processing method can be applied not only to the sparse filters but also to the feature maps of sparse CNN models to further optimize sparse convolutions. From a framework and tools point of view, the lessons learned thus far in using OpenVINO and the DLA development suite will be invaluable in our effort to enhance the DLA primitives and architecture to support emerging DNN models and applications.

Finally, the results from this study, as exemplified by the results presented in Section V, provide an excellent foundation for more extensive design space exploration going forward to investigate various architectural, model, and tool trade-offs on performance and other important metrics such as power and cost.

References

[1] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich *et al.*, "Going deeper with convolutions. arxiv 2014," *arXiv preprint arXiv:1409.4842*, vol. 1409, 2014.

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[7] E. Nurvitadhi, S. Subhaschandra, G. Boudoukh, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, and et al., "Can fpgas beat gpus in accelerating next-generation deep neural networks?" *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA 17*, 2017.

[8] I. K. Ganusov, M. A. Iyer, N. Cheng, and A. Meisler, "Agilex™ generation of intel® fpgas," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, pp. 1–26.

[9] D. Wang, J. An, and K. Xu, "PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks," *arXiv e-prints*, p. arXiv:1611.02450, Nov 2016.

[10] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 7, p. P07027, Jul 2018.

[11] DeePhi, "Deephi dnndk," http://www.deephi.com/technology/dnndk.

[12] Intel, "Openvino toolkit," https://software.intel.com/en-us/openvino-toolkit.

[13] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[14] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[16] R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, pp. 947–951, Jun. 2000.

[17] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model compression and acceleration for deep neural networks: The principles, progress, and challenges," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 126–136, 2018.

[18] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," *arXiv preprint arXiv:1608.04493*, 2016.

[19] S. F. Reddaway, "Dap—a distributed array processor," in *Proceedings of the 1st annual symposium on Computer architecture*, 1973, pp. 61–65.

[20] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[21] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[22] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[23] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[24] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. OConnell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, and G. R. Chiu, "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration," *arXiv e-prints*, p. arXiv:1807.06434, Jul 2018.

[25] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 769–780.

[26] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *International Conference on Computational Science*. Springer, 2005, pp. 99–106.

[27] S. L. Notes, "Coordinate format (coo)," 2018.

[28] Y. Saad, *Iterative methods for sparse linear systems*. siam, 2003, vol. 82.

[29] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. Siam, 1994, vol. 43.

[30] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 15–28.

[31] M. Adhiwiyogo, R. D'Souza, S. Leibson, and R. Shah, "Pushing ai boundaries with scalable compute-focused fpgas."

[32] A. Demidovskij, A. Tugaryov, A. Suvorov, Y. Tarkan, M. Fatekhov, I. Salnikov, A. Kashchikhin, V. Golubenko, G. Dedyukhina, A. Alborova *et al.*, "Openvino deep learning workbench: A platform for model optimization, analysis and deployment," in *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2020, pp. 661–668.

[33] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.

[34] H. Vanholder, "Efficient inference with tensorrt," 2016.