Towards Efficient Neural Network Model Parallelism on Multi-FPGA Platforms

David Rodríguez Agut
Universitat Politècnica de València
Valencia, Spain

Rafael Tornero
Universitat Politècnica de València
Valencia, Spain

Josè Flich
Universitat Politècnica de València
Valencia, Spain

***Abstract*—Nowadays, convolutional neural networks (CNN) are common in a wide range of applications. Their high accuracy and efficiency contrast with their computing requirements, leading to the search for efficient hardware platforms. FPGAs are suitable due to their flexibility, energy efficiency and low latency. However, the ever increasing complexity of CNNs demands higher capacity devices, forcing the need for multi-FPGA platforms. In this paper, we present a multi-FPGA platform with distributed shared memory support for the inference of CNNs. Our solution, in contrast with previous works, enables combining different model parallelism strategies applied to CNNs, thanks to the distributed shared memory support. For a four FPGA setting, the platform reduces the execution time of 2D convolutions by a factor of 3.95 when compared to single FPGA. The inference of standard CNN models is improved by factors ranging 3.63-3.87.**

***Index Terms*—Multi-FPGA, Neural Networks, Deep Learning**

I. Introduction

In recent years, FPGAs have been gaining popularity not only as prototyping platforms but also as hardware accelerators. One of the fields where FPGAs have been explored, is Deep Neural Networks (DNN). DNNs are broadly used in a vast range of applications such as autonomous driving, image classification, natural language processing, computer vision, etc.

Among DNNs, convolutional neural networks (CNNs) stand out over the rest due to their great accuracy on many tasks though in exchange for a computational cost. CNNs are mainly composed of convolutional layers, each being a high-dimensional convolution operator. FPGAs stand out due to their low latency and reduced power consumption compared to CPUs/GPUs and flexibility compared to ASICs [1]–[3]. Although there has been work done on the training of neural networks with FPGAs [4], their use has been focused on inference with fixed-point or integer-based arithmetic where they can achieve lower latency and energy consumption.

Originally, most FPGA-based proposals target single FPGA designs [5]–[7]. However, with the increase in size of CNNs, and the prospect of models getting bigger by the time, a single FPGA device falls short with the increasing requirements in terms of resources. To overcome the resource limitation, and at the same time aiming to obtain better performance, in the last couple of years multi-FPGA platforms have been explored as an option. But with the use of multi-FPGA platforms new challenges arise, such as how to efficiently use all the resources available, or how to increase overall performance without affecting latency or energy consumption.

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558 (eFlows4HPC project). The JU receives support from the European Union's Horizon 2020 research and innovation programme, and Spain, Germany, France, Italy, Poland, Switzerland, and Norway.

With FPGAs two approaches can be followed. First, the model can be synthesised and optimised. This leads to the best performance achievable as the weights of the convolution filters can be encoded on chip using BRAMs. However, running different models requires a reprogramming step which usually takes hundreds of milliseconds. In addition, large models may not fit on-chip. Second, a generic accelerator (kernel) design can be implemented and programmed from the host, and iterated over in order to infer a complete model. The accelerator can be scaled to the FPGA resources and does not need to be reprogrammed to run different models.

When targeting multiple FPGAs, external memory allocation is critical. Each accelerator has access to its own local memory but to exchange data explicit memory transfers are needed. To overcome this, we propose a novel multi-FPGA platform for the effective execution of inference processes of CNN models. The approach relies on the distributed shared memory programming paradigm. Our platform implements a shared memory address space distributed across all FPGA boards, following a NUMA architecture. Thus, explicit memory transfers between FPGAs are avoided. On each FPGA we implement a generic accelerator which can be programmed to work on different partitioning schemes for a set of specific neural network layers. Therefore, several accelerators can be programmed to run the same set of layers concurrently and cooperatively. By the use of the shared memory address space, we allow different partitioning schemes to be used concurrently as well. This is key to enable close ideal speed-ups.

II. Related Work

In recent years several works have explored the concept of connecting several FPGAs with a certain topology to execute DNNs. In [8] authors propose a pipelined FPGA cluster to increase the throughput of CNN applications through the use of a dynamic programming algorithm to map CNN layers on the FPGAs. In their solution they use seven FPGAs, one used as controller, forming a ring topology. Each FPGA has a customised accelerator for a specific set of layers. As a result, the design lacks flexibility since the FPGAs are highly coupled with each other, so to execute different CNNs the FPGAs need to be reconfigured each time. Extending their original work in

[9] they propose a dynamic algorithm for mapping DNNs in asymmetric multi-FPGA platforms. In both works, they aim to map a DNN model on a multi-FPGA platform layer by layer to increase throughput. Following the same approach in [10]–[14] authors also propose different techniques and algorithms to efficiently map a DNN in a multi-FPGA platform.

Other works follow a different approach. In [15] a framework to partition CNNs in FPGA clusters is proposed. The focus is on reducing latency by employing different layer partition schemes, as well as by alleviating off-chip memory usage by making use of the inter-FPGA links. However, due to the implementation of their solution, employing different partition schemes between layers entails unavoidable data movement which in consequence is detrimental to their solution. Therefore, they are bound to use the same partition strategy between all the FPGAs so as not to affect latency as a result of explicit data exchange between FPGAs. Contrary to this, in this paper, we allow the combination of different partition schemes. Other multi-FPGA papers aim at reducing inference time in RNNs. In [16], [17] the Microsoft Brainwave project aims at reducing latency by pining weights on the FPGAs. In [18] authors also focus on reducing latency in RNNs by partitioning the layers and pipelining the computation.

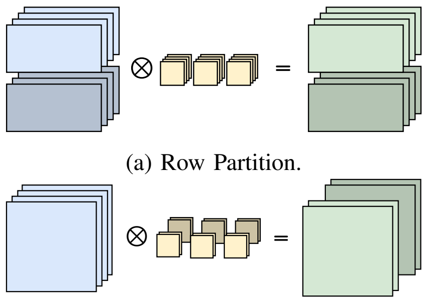
III. Background

The convolution operation is defined as $Input(I \times HI \times WI) \circledast Filter(O \times I \times KH \times KW) = Output(O \times HO \times WO)$. The input is defined as a three dimensional tensor $I \times HI \times WI$ where I represents the number of input channels (or feature maps) and HI and WI represent the height and width of each channel, respectively. Similarly, the convolution operation produces a three dimensional tensor output $O \times HO \times WO$. Typically, each input-output pair has a $KH \times KW$ filter. Thus, $O \times I$ filters are used in a convolution. The filters are applied on every $KH \times KW$ subset of inputs and are shifted vertically and horizontally over the input channels.

A convolution can be divided in partitions that run in parallel, leading to different distribution schemes of input activations, weights and feature maps. Mainly, we can apply batch, row, column, output channel and input channel partition schemes. Batch partition is mainly used in training processes. In real time inference processes batch size is set to one.

In (input) row partition (IRP, Fig. 1a), all input channels are split in several partitions, each with a disjoint set of rows. Each partition performs the convolution using the same filters, therefore filters are shared between partitions. Each convolution on each partition produces a subset of output rows for all output channels. Depending on the horizontal stride of the convolution some rows need to be shared between adjacent partitions, therefore, not being a perfect partitioning scheme. IRP requires that input data, although partitioned, be accessed by different FPGA accelerators. In a non-shared memory system this will entail explicit data copies. In column partition, channels are partitioned in sets of columns instead of rows. This scheme has the same side effects as row partition.

With output channel partition (OCP, Fig. 1b), the output feature maps are partitioned/computed in parallel. Filters are partitioned since they belong to different output channels. However, input data is shared and thus, every accelerator needs to read the same data. Finally, in input channel partition, the input is partitioned in sets of channels and for each partition the complete set of output channels is computed. Similarly to OCP, filters are partitioned by input channel. However, the output produced by each partition must be compounded by an additional layer to add all partitions together.



(a) Row Partition.

(b) Output Channel Partition.

Fig. 1: Layer Partition schemes.

IV. Shared Memory Design

Our baseline design (Fig. 2) features four Xilinx Kintex Ultrascale FPGAs interconnected following a bidirectional ring topology. Each FPGA board has two QSFP+ modules and four optical cables for communication with its two neighbour FPGA boards. At the design level, Chip2Chip and Aurora IP cores from Xilinx have been used for the communication between FPGAs. These IP cores allow board-to-board communication following the AXI4 protocol. Additionally, each FPGA board includes a 2GB DDR4 memory module and its corresponding memory controller. This controller handles local memory accesses and has not been modified in our design. Furthermore, one FPGA is connected to the host system through PCIe via the PCIe IP core. Each FPGA board includes also an accelerator. All the components within the design are interconnected and configured via an AXI interconnect infrastructure. As a result, for the host application the whole design appears as a single platform with four accelerators and 8GB of memory.

A. Distributed Shared Memory Support

For distributed shared memory support, each FPGA includes a hierarchy of AXI interconnects (SmartConnect IP). Four AXI modules are used on $FPGA_1$, $FPGA_2$ and $FPGA_3$; whereas six are used in $FPGA_0$. Up to three modules route incoming traffic either from a neighbour FPGA, which can access the local memory or being routed to another FPGA (using this FPGA as in-transit), from the local accelerator, or from the PCI express module in case of $FPGA_0$. These AXI modules have different outputs depending on the communication patterns allowed. Each output can deliver the data directly to the final destination: local memory, or arrive to a new AXI module that delivers the data to the corresponding neighbour FPGA device. The interconnection between AXI modules is not a fully connected topology as some links are not implemented.

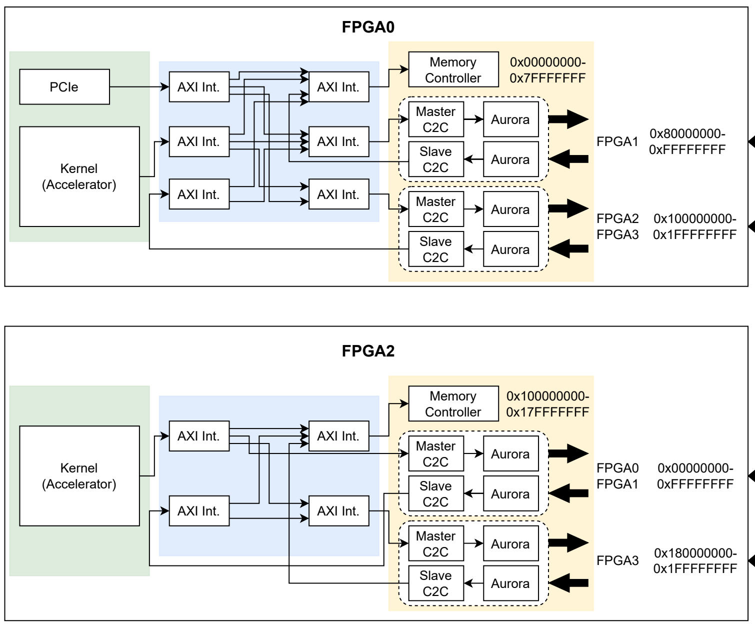


Fig. 2: Multi-FPGA architecture with distributed shared memory

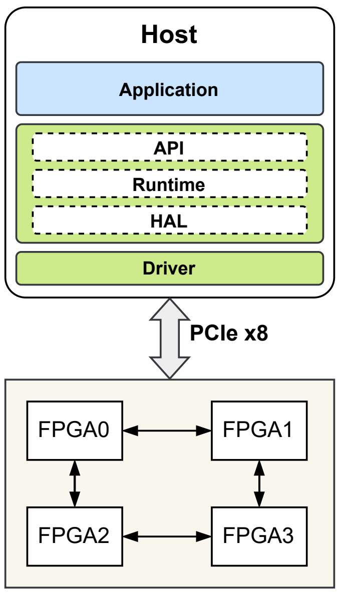


Fig. 3: Host layer hierarchy

All the modules have been configured to support distributed shared memory. Every accelerator on any FPGA will be able to access all the memories using the same memory addressing space. Table I shows the configuration of the AXI interfaces on each FPGA with the range of addresses and the paths used to access each memory by all FPGA devices. Physically, FPGAs are connected following a bidirectional ring. However, internally, FPGAs are logically grouped by rows. Each FPGA can access its direct neighbours DDR through the inter-FPGA links. But to access a DDR memory located on a non-neighbour FPGA, it has to go through its neighbour in the other row. For instance, $FPGA_0$ will have to go through $FPGA_2$ in order to access DDR_3. This means also that $FPGA_0$ and $FPGA_2$ will end up sharing the link between $FPGA_2$ and $FPGA_3$ if both try to access DDR_3 at the same time.

FPGA links are configured to work at an effective read and write bandwidth of 2.2 GB/s duplex mode. The bandwidth is limited by the QSFP+ modules which support a maximum transfer rate of 10 Gb/s per lane, whereas FPGA transceivers support data rates up to 16.375 Gb/s, and by the Chip2Chip IP core that in tandem with Aurora can only be configured to use three lanes maximum. Therefore links are configured to use three lanes each working at 10 Gb/s.

B. Runtime software

The platform is exposed to the host as a memory mapped PCIe device with two memory regions: one for controlling the accelerators in each FPGA, and one for mapping the different device memory banks (DDRs) through a unified address space. In order to provide access to this platform we have developed an OpenCL-like runtime software library, inspired by the Xilinx OpenCL software stack and composed of three layers: an API, a runtime and a hardware abstraction layer (HAL).

Fig. 3 illustrates the hierarchy of layers in the library. From top to bottom, the API presents two main abstractions to the host application, named device and buffer. These abstractions are similar to the respective abstractions in the OpenCL specification. In the platform initialisation process an instance of the device abstraction is created for each device in the platform. Through this abstraction, the host application developer can control the kernels programmed in the physical device the instance has been attached to. The buffer abstraction represents a memory area located in device memory, and the API provides high level functions to perform efficient memory-to-memory data transfers from the host system memory to those buffers allocated in some of the device memory banks. The runtime layer sits between the API and the HAL. It provides the required services for implementing the API, taking care of the overall platform memory management in a flexible way. The HAL provides a high level interface that hides the physical platform details to the runtime layer and it relies on the Xilinx XDMA driver [19] to implement the low-level communication details to the hardware platform. It provides two access modes to the hardware. First, it allows to read and write 32- or 64-bit data registers using 32-bit register addresses. Second, it allows to program the platform DMA to perform efficient memory transfers between host and device memory.

C. FPGA-FPGA Link Evaluation

Once we have defined our platform with distributed shared memory capabilities, we need to define the allocation and mapping strategy of both data and the accelerator workload. Notice that depending on the location of the data, accelerators may require to access non-local memories and could even congest some links. Before defining such mapping policy we need first to evaluate the bandwidth attainable by FPGA-to-FPGA links and to analyse the read and write requirements of the different types of data used in an inference process.

To check the impact in performance of the FPGA-FPGA links, Table II shows the relative performance achieved when

Address range	$FPGA_0$	$FPGA_1$	$FPGA_2$	$FPGA_3$
0x000000000 0x07FFFFFFF	Local	$FPGA_1 \rightarrow FPGA_0$	$FPGA_2 \rightarrow FPGA_0$	$FPGA_3 \rightarrow FPGA_1 \rightarrow FPGA_0$
0x080000000 0x0FFFFFFFF	$FPGA_0 \rightarrow FPGA_1$	Local	$FPGA_2 \rightarrow FPGA_0 \rightarrow FPGA_1$	$FPGA_3 \rightarrow FPGA_1$
0x100000000 0x17FFFFFFF	$FPGA_0 \rightarrow FPGA_2$	$FPGA_1 \rightarrow FPGA_3 \rightarrow FPGA_2$	Local	$FPGA_3 \rightarrow FPGA_2$
0x180000000 0x1FFFFFFFF	$FPGA_0 \rightarrow FPGA_2 \rightarrow FPGA_3$	$FPGA_1 \rightarrow FPGA_3$	$FPGA_2 \rightarrow FPGA_3$	Local

TABLE I: Path configuration for the multi-FPGA platform.

running a 2D convolution with a $256 \times 256 \times 256$ input. The accelerator has been tested running on each FPGA (K_i when running on $FPGA_i$) and data has been located (input, output, and filters) on each DDR memory (DDR_i). Remote access to neighbour DDRs has a negligible impact in performance compared to local memory, thus both being comparable. However, accessing a remote non-neighbour DDR (e.g. $K_0 \leftrightarrow DDR_3$) entails a penalty of around 9-11%. Thus, access to remote non-neighbour memory should be minimised and controlled.

	K_0	K_1	K_2	K_3
DDR_0	1.00	0.99	0.99	0.90
DDR_1	0.99	1.00	0.90	0.99
DDR_2	0.99	0.91	1.00	0.99
DDR_3	0.88	0.99	0.99	1.00

TABLE II: Relative performance of a $256 \times 256 \times 256$ convolution on different FPGA devices and data location.

Another important aspect to analyse are the read and write requirements of the accelerator for the different types of data (input, filters and outputs). Indeed, convolution operations have a large arithmetic intensity since weights are reused over the entire input. This suggests that weights read bandwidth requirements are much lower than input and output read and write bandwidth requirements. In order to confirm this, we have reproduced the previous experiment but in this case we have allocated input and output data on the local DDR memory where the accelerator is running. Weights are placed on the most distant DDR (e.g. k_0 running and accessing input and output data from DDR_0 and weights at DDR_3). The performance of the accelerator when performing the convolution equals the optimal performance achieved when all the data is local. This means that weights location do not affect final performance. As an additional experiment we placed input and output data on remote DDR and weights on local DDR. We achieved the most significant impact approaching 0.9 of relative performance.

D. Mapping Algorithm

We devised a mapping algorithm (Listing 1) to determine which kernels will run and which work they will perform (rows to read from input and output channels to compute). The algorithm uses either output channel partition (OCP) and input row partition (IRP) schemes at the same time (we refer to it as Hybrid Partition, HP) or just OCP. The algorithm is computed offline and before inference is performed. It is called for every layer of the model.

The algorithm takes as threshold the number of rows (row_{th}) to decide whether HP or OCP is used (line 2). The threshold has been obtained experimentally from the platform and depends on the data type used and the link bandwidth. For our tests it is set to 56. The number of output channels determines how many accelerators will be launched (line 1). We assume enough output channels are provided for dividing the workload between two accelerators in the HP case (lines 5-6).

Algorithm 1 Mapping algorithm.

```
1:  $n_k \leftarrow min(4, \frac{O}{CPO})$
2:  **if** $H \geq row_{th}$ & $n_k = 4$ **then**                ▷ Hybrid Partition
3:      $rows_{\{k_0,k_1\}} \leftarrow 0 \dots \frac{H}{2} - 1$
4:      $rows_{\{k_2,k_3\}} \leftarrow \frac{H}{2} \dots H - 1$
5:      $OC_{\{k_0,k_2\}} \leftarrow 0 \dots \frac{O}{2} - 1$
6:      $OC_{\{k_1,k_3\}} \leftarrow \frac{O}{2} \dots O - 1$
7:      ASSIGN BUFFER(...)                                  ▷ Call to API
8:      LAUNCH KERNEL($k_{th}, rows, OC, \dots$)
9:  **else**                                       ▷ Output Channel Partition
10:     $OC_{\{k_{th}\}} \leftarrow \frac{O}{n_k} \times k_{th} \dots \frac{O}{n_k} \times (k_{th}+1) - 1$
11:     ASSIGN BUFFER(...)
12:     LAUNCH KERNEL($k_{th}, OC, \dots$)
13: **end if**
```

When assuming HP, the input (lines 3-4) and output (lines 5-6) configuration of each accelerator is different (input data can be distributed in different DDR memories). In this case, the runtime uses DDR_0 and DDR_2 for storing those buffers with the input data being halved between them. So, each accelerator will process $I \times \frac{H}{2} \times W$ pixels of the input data, and will compute half of the output channels for that part of the input data: $\frac{O}{2} \times \frac{H}{2} \times W$. Notice that DDR assignment is performed by the runtime.

Additionally, K_0 and K_1 will work in tandem with DDR_0 as well as K_2 and K_3 with DDR_2. Since accessing filters and bias has a negligible impact on performance, as commented before, they will be in DDR_0. On the other hand, when assuming OCP, the algorithm will map $\frac{O}{n_k}$ output channels to each accelerator. Notice n_k determines the number of accelerators to use. Weights are not shared, therefore they can be distributed among all the memories. Function ASSIGN_BUFFER (lines 7 and 11) configures the buffers for outputs and weights and assigns them to DDR memories.

Fig. 4 shows the mapping produced for the VGG16 model as an example. Here we can see how HP is used in the first seven convolution layers. The remaining layers only use OCP. Notice how the four kernels run concurrently for all the layers, achieving maximum kernel utilisation. In the first seven layers two groups of kernels work on the same output channels but each computing different output rows. Data is split by the runtime between DDR_0 and DDR_2. Indeed, the kernels are the ones that write in the correct DDR memory. As an example, in

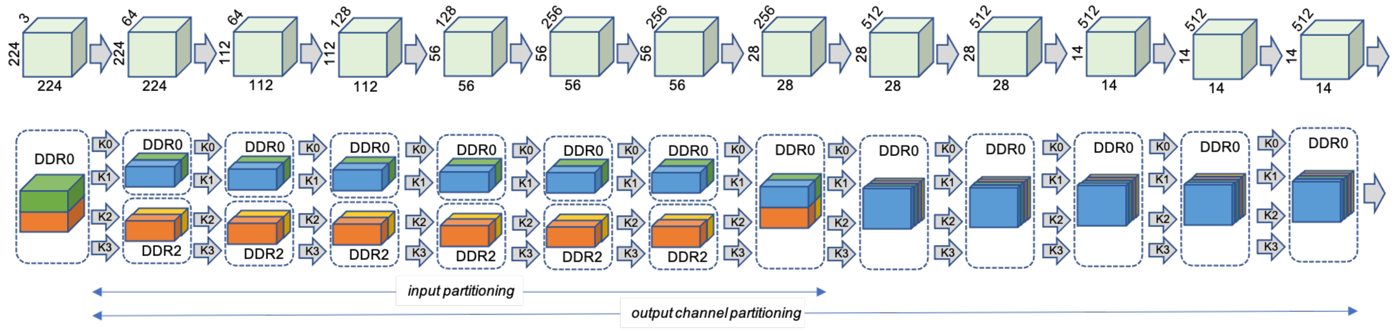


Fig. 4: VGG Mapping on the multi-FPGA platform.

the first layer all the kernels read the input data from DDR_0 but two of the kernels write the results on DDR_2. In the following layers this split is already taken into consideration, so K_2 and K_3 operate on DDR_2. Notice that in the seventh layer all the kernels write on DDR_0 since the next layer does not use HP. From that layer on, all the kernels work concurrently on different output channel intervals, thus exploiting only OCP.

V. Design Evaluation

To evaluate our platform we run multiple inferences and compute the average time for each FPGA configuration. Inference time has been measured from the host. To carry out the tests we used the HLSinf [20] accelerator configured to use FP32 data type. Although it is advisable to use fixed point data types in FPGAs, the accelerator achieves better performance with FP32. Additionally, FP32 allows us to stress the inter-FPGA links. For the host-side application we used the EDDL library [21] to train and infer neural network models, similarly to TensorFlow. VGG16, Hourglass neural network for human pose estimation (HG) and Residual network (Resnet50) models are used. We compare the results of speed-up using output channel partition (OCP) with all the accelerators accessing the same DDR, with hybrid partition (HP) and a combination of both strategies using the mapping algorithm described in the previous section. Input data is $3 \times 224 \times 224$ pixels.

First, we analyse the efficiency when using OCP and remote memory access. In this test each accelerator has its weights and bias in its local memory (K_i on DDR_i) whereas data (input and output) is kept at DDR_0. Thus, all accelerators access DDR_0 at the same time. Fig. 5 shows the speed-up achieved when compared to single FPGA for the three analysed CNN models. OCP results correspond to the dotted line. As can be seen, with two FPGAs the speed-up is close to linear, however with four FPGAs the platform fails to achieve optimal performance. As discussed, the cause for this drop is the overload in one of the inter-FPGA links, as well as the access to non-neighbour memory without any mechanism to control it.

In order to avoid this loss, we use hybrid partitioning (HP) by combining IRP with OCP. This way instead of having the input data in one DDR that is accessed by all the accelerators, the input data in distributed in two DDRs (DDR_0 and DDR_2). As before, weights and bias are in the local memory of each accelerator. The speed-up achieved with HP is shown in Figure 5 (dashed line). In the VGG and HG models the results have significantly improved. However, in the Resnet model the use of HP translated in negligible gains. Even though we tackled the cause for the loss in performance, the platform still fails at achieving optimal results. Also, VGG and HG still do not achieve acceptable performance values.

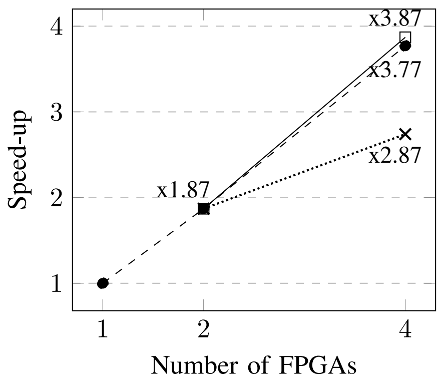
To further increase performance we have analysed the speed-up obtained with both strategies. Table III shows the speed-up achieved for each layer in the VGG model when using either OCP or HP. Now, some layers behave much better with HP whereas others perform better with only OCP. The layers that run better with OCP correspond to layers with a large number of output channels and not many rows per channel, whereas layers with large number of rows per channel behave better with HP. This motivates for the algorithm we introduced.

Configuration	Speed-up OCP	Speed-up HP	OCP vs HP
3x224x224	2,80	3,70	0,90
64x224x224	1,75	3,94	2,19
64x112x112	1,78	3,88	2,11
128x112x112	2,04	3,90	1,86
128x56x56	2,45	3,70	1,25
256x56x56	2,45	3,84	1,39
256x56x56	3,76	3,84	0,08
256x28x28	3,85	3,70	-0,16
512x28x28	3,87	3,73	-0,14
512x28x28	3,87	3,72	-0,14
512x14x14	3,91	3,43	-0,48
512x14x14	3,91	3,43	-0,48
512x14x14	3,81	3,43	-0,38

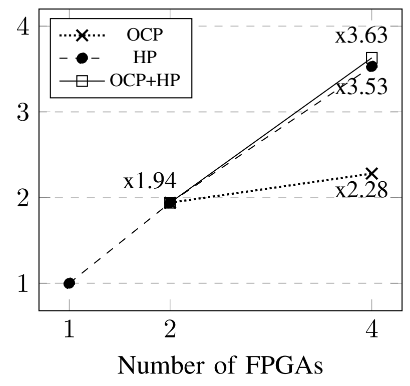
TABLE III: Performance gain with OCP and HP.

Thus, we now apply the mapping algorithm. Some layers are computed with HP and others with OCP. Data buffers are also strategically put either in DDR_0 or split in DDR_0 and DDR_2 based on the mapping algorithm. Fig. 5 shows the speed-up achieved when using the mapping algorithm (dense line). With the use of both partition strategies we further increase performance with four FPGAs. With VGG and HG we gain a 10% in performance whereas with ResNet the gain is 66%.

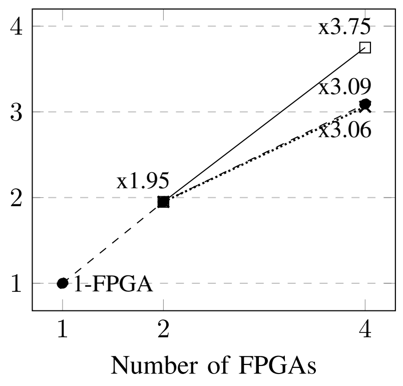
As we can see from the results, there is still some margin for improvements. Indeed, VGG, HG, and ResNet achieve 3.87, 3.63, and 3.75 speed-up factors, respectively. After measuring the link bandwidths and the memory bandwidth requirements for the different models, we confirmed that the platform was not introducing any bottleneck. Thus, we focus on the efficiency of



(a) VGG.



(b) HG.



(c) Resnet.

Fig. 5: Performance results achieved for varying number of FPGAs and partitioning schemes. Different NN models used.

the accelerator. Table IV shows the performance achieved by four different synthetic models made of a single 2D convolution layer, each model with a different size of convolution. We run the inference of those models in one FPGA and compare with the theoretical execution time to obtain the efficiency. The theoretical execution time is obtained by the formula: $H \times W \times \frac{I}{CPI} \times \frac{O}{CPO}$ cycles.

Configuration	Efficiency	Configuration	Efficiency
16x16x16x16	0.39	128x128x128x128	0.97
64x64x64x64	0.93	256x256x256x256	0.98

TABLE IV: Accelerator efficiency with different convolutions.

Thus, the reason for the loss in performance is due to some internal inefficiencies of the accelerator. For small convolutions the accelerator does not achieve maximum performance. Contrary to this, the platform manages large convolutions with an efficiency factor close to the ideal factor.

VI. Conclusion

We presented a multi-FPGA platform with shared distributed memory for the inference of CNNs. With the support for shared distributed memory, we combine different partition strategies to achieve close to linear speed-up with up to four FPGAs. Furthermore, we have devised a preliminary mapping algorithm that takes into account the parameters of each layer of a given model and applies different partition strategies to obtain optimal performance. The platform has enough communication resources to increase the number of FPGAs, and the mapping algorithm and host library can be improved to achieve higher performance and support complex workloads.

References

[1] E. Nurvitadhi *et al.*, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 Int. Conf. on Field-Programmable Technology*. Xi'an, China: IEEE, Dec. 2016, pp. 77–84.

[2] J. Qiu *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *Proc. of the 2016 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*. ACM, Feb. 2016, pp. 26–35.

[3] J. Cong *et al.*, "Understanding Performance Differences of FPGAs and GPUs," in *2018 IEEE 26th Annual Int. Symp. on Field-Programmable Custom Computing Machines*. Boulder, CO, USA: IEEE, Apr. 2018.

[4] T. Geng *et al.*, "A Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Work and Weight Load Balancing," in *2018 28th Int. Conf. on Field Programmable Logic and Applications (FPL)*. Dublin, Ireland: IEEE, Aug. 2018, pp. 394–3944.

[5] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proc. of the 2015 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[6] Y. Ma *et al.*, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *Proc. of the 2017 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*. Monterey California USA: ACM, Feb. 2017, pp. 45–54.

[7] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *Proc. of the 2016 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*. Monterey California USA: ACM, Feb. 2016, pp. 16–25.

[8] C. Zhang *et al.*, "Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster," in *Proc. of the 2016 Int. Symp. on Low Power Electronics and Design*. San Francisco Airport CA USA: ACM, Aug. 2016, pp. 326–331.

[9] W. Zhang *et al.*, "An Efficient Mapping Approach to Large-Scale DNNs on Multi-FPGA Architectures," in *2019 Design, Automation & Test in Europe Conf. & Exhibition*. IEEE, Mar. 2019, pp. 1241–1244.

[10] J. Shen *et al.*, "Scale-out Acceleration for 3D CNN-based Lung Nodule Segmentation on a Multi-FPGA System," in *Proc. of the 56th Annual Design Automation Conf. 2019*. Las Vegas NV USA: ACM, Jun. 2019.

[11] S. Biookaghazadeh *et al.*, "Toward Multi-FPGA Acceleration of the Neural Networks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 2, pp. 1–23, Apr. 2021.

[12] J. Shan *et al.*, "CNN-on-AWS: Efficient Allocation of Multikernel Applications on Multi-FPGA Platforms," *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 2, Feb. 2021.

[13] R. Li *et al.*, "Improving CNN performance on FPGA clusters through topology exploration," in *Proc. of the 36th Annual ACM Symp. on Applied Computing*. Virtual Event Republic of Korea: ACM, Mar. 2021.

[14] T. Alonso *et al.*, "Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 2, Jun. 2022.

[15] W. Jiang *et al.*, "Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, pp. 1–23, Oct. 2019.

[16] E. Chung *et al.*, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, Mar. 2018.

[17] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual Int. Symp. on Computer Architecture (ISCA)*. Los Angeles, CA: IEEE, Jun. 2018, pp. 1–14.

[18] Y. Sun *et al.*, "FiC-RNN: A Multi-FPGA Acceleration Framework for Deep Recurrent Neural Networks," *IEICE Transactions on Information and Systems*, vol. E103.D, no. 12, pp. 2457–2462, Dec. 2020.

[19] Xilinx, "Xilinx DMA IP Reference drivers," https://github.com/Xilinx/dma_ip_drivers.

[20] J. Flich *et al.*, "Efficient Inference Of Image-Based Neural Network Models In Reconfigurable Systems With Pruning And Quantization," in *2022 IEEE International Conference on Image Processing (ICIP)*. Bordeaux, France: IEEE, Oct. 2022, pp. 2491–2495.

[21] H. D. project, "European Distributed Deep Learning (EDDL) Library," https://github.com/deephealthproject/eddl.