Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning

TOBIAS ALONSO, Universidad Autónoma de Madrid
LUCIAN PETRICA, Xilinx Research, Ireland
MARIO RUIZ, Xilinx University Program, Ireland
JAKOBA PETRI-KOENIG, Delft University of Technology, Netherlands
YAMAN UMUROGLU, Xilinx Research, Ireland
IOANNIS STAMELOS and ELIAS KOROMILAS, InAccel, US
MICHAELA BLOTT, Xilinx Research, Ireland
KEES VISSERS, Xilinx Research, US

Customized compute acceleration in the datacenter is key to the wider roll-out of applications based on deep neural network (DNN) inference. In this article, we investigate how to maximize the performance and scalability of field-programmable gate array (FPGA)-based pipeline dataflow DNN inference accelerators (DFAs) automatically on computing infrastructures consisting of multi-die, network-connected FPGAs. We present Elastic-DF, a novel resource partitioning tool and associated FPGA runtime infrastructure that integrates with the DNN compiler FINN. Elastic-DF allocates FPGA resources to DNN layers and layers to individual FPGA dies to maximize the total performance of the multi-FPGA system. In the resulting Elastic-DF mapping, the accelerator may be instantiated multiple times, and each instance may be segmented across multiple FPGAs transparently, whereby the segments communicate peer-to-peer through 100 Gbps Ethernet FPGA infrastructure, without host involvement. When applied to ResNet-50, Elastic-DF provides a 44% latency decrease on Alveo U280. For MobileNetV1 on Alveo U200 and U280, Elastic-DF enables a 78% throughput increase, eliminating the performance difference between these cards and the larger Alveo U250. Elastic-DF also increases operating frequency in all our experiments, on average by over 20%. Elastic-DF therefore increases performance portability between different sizes of FPGA and increases the critical throughput per cost metric of datacenter inference.

CCS Concepts: • **Computer systems organization → Distributed architectures;**

Additional Key Words and Phrases: Deep neural networks, partitioning, distributed inference

Authors' addresses: T. Alonso, Autonomous University of Madrid, Madrid, Spain; email: tobias.alonso@uam.es; L. Petrica, Y. Umuroglu, and M. Blott, Xilinx Research, Dublin, Ireland; emails: {lucianp, yamanu, michaela.blott}@xilinx.com; M. Ruiz, Xilinx University Program, Dublin, Ireland; email: mruiznog@xilinx.com; J. Petri-Koenig, Delft University of Technology, Delft, Netherlands; email: J.Petri-Koenig@tudelft.nl; I. Stamelos and E. Koromilas, InAccel, US; K. Vissers, Xilinx Research, San José, US; email: kees.vissers@xilinx.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1936-7406/2021/12-ART15 $15.00
https://doi.org/10.1145/3470567

ACM Reference format:
Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. 2021. Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 15 (December 2021), 34 pages.
https://doi.org/10.1145/3470567

1 INTRODUCTION

Compute acceleration in the **datacenter (DC)** is seen as an enabler for a wide range of new machine learning-based applications, which rely on **deep neural network (DNN)** inference [15]. As DNN inference-based applications mature and enter production, an increasing focus is placed on the efficiency of computation and flexibility of deployment of those applications. In the first phase of deployment, the application operator must map their DNN model to the available DC accelerator hardware, subject to certain application-specific latency constraints. Subsequently, the inference application is deployed to (usually many) instances of the cloud accelerator, rented at fixed cost per hour, and hence the application operator desires to maximize the number of DNN queries per second (and therefore per unit of cost).

For the DC operator, whose incentive is to maximize profit, the essential metrics are the cost of operating the accelerator, which depends on the power dissipation of the accelerator and surrounding system (CPUs, memory, storage), and the achievable revenue stream per accelerator. Power efficiency considerations have led to the development of a wide range of specialized deep learning accelerators such as Google's TPU [27], Amazon's Inferentia chip, and Huawei's Ascend 910 by the hyperscalers themselves, replacing GPUs. These specialized compute architectures reduce dissipated power for a given number of DNN queries per second. Revenue stream optimization has led to an increasing trend toward accelerator multi-tenancy, which DC operators can leverage to extract multiple revenue streams per accelerator chip, and dis-aggregation of acceleration in the datacenter, which enables flexible coupling of accelerators to CPUs, memory and storage for specific user requirements, reducing waste from idle CPU/accelerator cycles or unused memory.

Recently **field-programmable gate arrays (FPGAs)** have also gained a foothold in the datacenter DNN inference acceleration market and appear well aligned with the above-mentioned trends. Unlike other DC compute accelerators, FPGAs enable complete customization of accelerator architecture, allowing computational logic to fit application precision, and memory hierarchies to fit application memory access patterns, thus minimizing the quantity of accesses to external memory and in theory achieving low-latency, power-efficient acceleration. Furthermore, FPGAs can connect directly to the DC network through dedicated network stacks implemented in FPGA fabric, minimizing communication latency between co-executing accelerators, reducing the total system power required to implement distributed applications, through the removal of CPU intermediation, and facilitating accelerator disaggregation. Finally, the multi-die structure of DC FPGAs creates the potential for multi-tenancy by allocating individual dies or groups thereof to each customer. However, making use of these features to maximize DC user and operator figures of metric requires the right choice of DNN inference accelerator architecture and an appropriate set of tools for its implementation.

We distinguish between two architectures for DNN inference acceleration on FPGA: **matrix of processing engines (MPE)** and a feed forward **dataflow (DF)**. MPE architectures, i.e., systolic-array matrix multipliers, execute instructions on fixed precision (typically 4/8bit) weights and activations fetched from external memory through deep-learning optimized memory hierarchies. The

generic structure of an MPE accelerator is reusable across different DNN layers or topologies, and is therefore more amenable to expert-guided optimization, which results in high-frequency implementations. However, the requirement for generality limits the amount of customization that can be applied to an MPE accelerator. Conversely, dataflow architectures leverage customization as a primary source of performance. To construct a **dataflow accelerator (DFA)**, we convert the DNN computational graph into a pipeline of FPGA accelerated operations which store all parameters they require to perform computation, e.g., weights and biases, in **on-chip memory (OCM)**. In this approach, the amount of data transferred across chip boundaries is minimized and therefore latency and power are reduced, but the size of the implementable accelerator is limited by the OCM of the target FPGA device. Second, custom FPGA circuitry must be generated for each DNN topology, typically through compilation of high-level C++ descriptions, limiting the opportunities for frequency optimization, especially on large multi-die FPGAs.

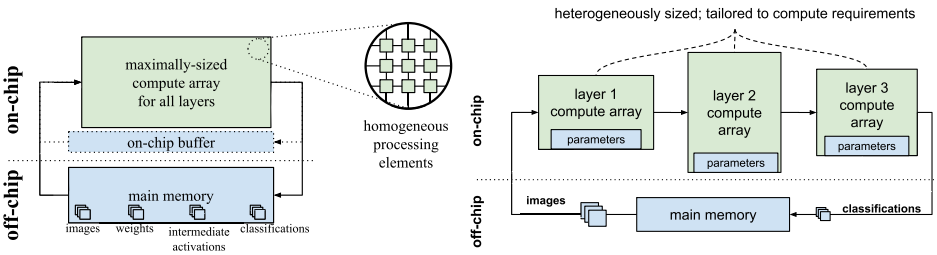
In this article, we focus on the implementation of design automation tools for DFAs in the DC context. We present Elastic-DF, an open-source partitioner and resource allocator which works in conjunction with the DFA compiler framework FINN [58], and peer-to-peer Ethernet infrastructure for direct communication between FPGAs. Elastic-DF enables the automatic model-parallel (MP) distribution of a DFA across multiple FPGAs in a software-transparent way, removing the OCM bottleneck of DF acceleration and paving the way for FPGA disaggregation. Furthermore, by combining partitioning and resource allocation, Elastic-DF increases DFA density. The DC user benefits from the increased density directly if they are able to fit more instances of the DFA into the FPGA, increasing the per-FPGA throughput. If this is not possible, then in a multi-tenant environment the DC user may require fewer FPGA dies (and therefore pay less) to implement their DFA, while the DC operator is free to leverage the remaining dies for other customers.

Our article makes the following specific contributions:

- We introduce an **Integer Linear Program (ILP)**-based partitioner and resource balancer, which is able to split a DFA across multiple dies or FPGAs, under communication throughput and FPGA congestion constraints, and VNx, a light-weight IP core that enables direct FPGA to FPGA communication via the UDP/IP protocol and 100 Gbps Ethernet. We open-source both of these tools [41, 45].
- We demonstrate that model-parallel multi-FPGA execution enables superlinear scaling of DFA performance in multi-FPGA systems, for both MobileNetV1 and ResNet-50. We show how constraints to Elastic-DF enable us to implement **software-transparent model parallelism (TMP)** and analyze how TMP compares with other forms of MP.
- We demonstrate the increase in DFA density via tight integration between partitioner and resource balancer. On selected boards and DNNs, we observe 78% increase in total FPGA inference throughput, compared to performing partitioning and balancing separately.
- We analyze the effect of partitioning on achievable DFA frequency. We argue that partitioning facilitates timing closure, in general, enabling over 20% improvement in operating frequency on average compared to non-partitioned designs built with Xilinx Vitis, helping to close the frequency gap between MPEs and DFAs. We open-source code exemplifying the application of Elastic-DF to FINN DNNs for frequency optimization [32].

2 BACKGROUND

In this section, we provide an overview of single-node and distributed DNN inference acceleration, focusing in particular on customized dataflow accelerators. We discuss how they can scale to meet throughput and latency or resource constraints and explain the associated toolflow, which includes the partitioner that is the main focus of this article.



(a) Matrix of Processing Elements (MPE). (b) Customized, Streaming Dataflow (DF).

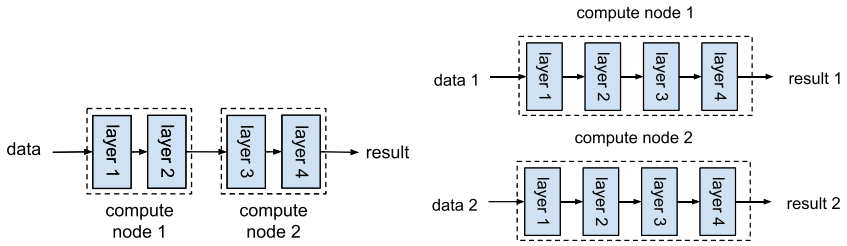
Fig. 1. MPE and DF acceleration paradigms.

2.1 Hardware Architectures for DNN Inference Acceleration

DNNs contain a multitude of **multiply-accumulate (MAC)** operations interspersed with nonlinearities, organized into *layers*. We refer to the reader to the survey by Sze et al. [53] for an overview of the computational aspects of DNNs and key techniques like sparsity and quantization to reduce the cost of computation. The inference computation for a DNN can be parallelized in different ways, which becomes evident if the inference computation is viewed as a dataflow graph. As long as the data dependencies are respected, the inference computation can be performed by multiple units operating along different axes of parallelism, including but not limited to: (1) different layers in a pipeline, (2) inputs in a batch, (3) channels and pixel positions for convolutions, (4) neurons and synapses for fully connected layers, and (5) bit positions within a single MAC for multi-bit arithmetic. Many accelerator architectures have emerged [53] that try to address the problem of parallelization in different ways, where the key challenge is: *How do we loop transform and unfold the algorithms to maximize data reuse and compute efficiency, minimize memory bottlenecks, limit power consumption while meeting latency requirements?* We focus on two prominent paradigms for designing single-FPGA DNN inference accelerators:

Matrix of Processing Elements (MPE)-style accelerators. A popular alternative [15, 16, 30, 64] for inference acceleration is to use a hardware architecture that offers a fixed degree of parallelism along one or more axes listed above, with a homogeneous array of parallel MAC units interconnected in a certain fashion and using a combination of on- and off-chip memory to store weights and activations. The movement between the memories and compute units is typically orchestrated by a sequence of instructions generated by a compiler: for each layer in the DNN, the input activations and weights for that layer are brought into on-chip memory, the computations are performed to produce output activations, which may need to be spilled into off-chip memory if the on-chip buffering is not sufficient. Afterwards, the output activations flow back into the accelerator for the next layer. As the same homogeneous array of PEs is used for computing each layer, we refer to this paradigm as ***Matrix of Processing Elements (MPE)***, and illustrate its general form in Figure 1(a). This paradigm allows the hardware architecture to be generated and optimized only once, and a wide variety of DNNs can be subsequently executed by compiling them onto this fixed architecture. This "one-size-fits-all" approach also comes with several drawbacks:

- Fixed parallelism along pre-determined axes may be under-utilized depending on the particular DNN topology; for instance, datapaths optimized for parallelizing dense convolutions will perform poorly for depthwise convolutions.
- The layer-by-layer data movement requires a high level of compiler and hardware optimization effort to achieve concurrent computation and communication, imposing limits on latency and efficiency due to overheads.



(a) Model parallel distribution. (b) Data parallel distribution.

Fig. 2. Two types of parallelism that enable scaling up an application to multiple accelerators.

- Supporting new layer types require changes to the fixed datapath, the hardware-software interface, and the compiler.

Some of these concerns can be addressed via runtime reconfiguration of the FPGA, but this introduces extra reconfiguration overhead in tens of milliseconds [6, 7] for large FPGAs, limiting its applicability for stringent low latency applications or small batch size requirements.

Customized, Streaming Dataflow (DF)-style accelerators. In this paradigm, a customized hardware architecture is specialized for a specific DNN topology, where each layer of the DNN is mapped to its own set of dedicated compute and memory resources as illustrated in Figure 1(b). Each layer's compute array is allocated compute resources proportional to the number of MACs needed by that layer, and communicates with neighboring compute arrays via on-chip FIFO channels. To minimize latency, each layer's compute array starts computing as soon as the previous one starts producing output. Prior works exploring this paradigm such as fpgaConvNet [60], FINN [58], FINN-R [5] and ReBNet [17] have demonstrated how creating a customized streaming dataflow architecture tailored to the requirements of each layer can reap more of the benefits of reconfigurable devices. The main drawback of this paradigm is resource limitations due to its spatial-processing nature: it may require a large amount of resources that may not fit into a single FPGA, which is one of the challenges addressed in this work.

2.2 Data- and Model-parallelism for Distributed DNN Inference

For both MPE and DF-style architectures, a single accelerator is rarely sufficient to provide the inference throughput required by datacenters at real-world scales. When this is the case, the solution is to scale up the inference process to multiple accelerators. This involves distributing the inference computation to network-connected nodes along the different axes of parallelism described in Section 2.1, in addition to the parallelism provided by the single accelerator. Readers can refer to Reference [25] for an in-depth description of possible approaches to distributed inference. In this work, we focus on data and model parallelism, illustrated in Figure 2.

In **data-parallel execution (DP)**, the input is subdivided into batches that are each processed by one of the available accelerators. Each accelerator executes the entire processing required to produce results from its allocated inputs. One consequence of this is that each worker must also be able to execute the entire DNN model. Furthermore, latency is directly proportional to the batch size as more multiple images have to be buffered before they can be processed. The **model-parallel approach (MP)** subdivides the DNN model into pieces allocated to each of the available accelerators, which process all the inputs to produce intermediate results, which they communicate to the next accelerator in the processing chain. Each accelerator stores only a fraction of the DNN model, but the amount of intermediate results communicated may be large and varies from accelerator to

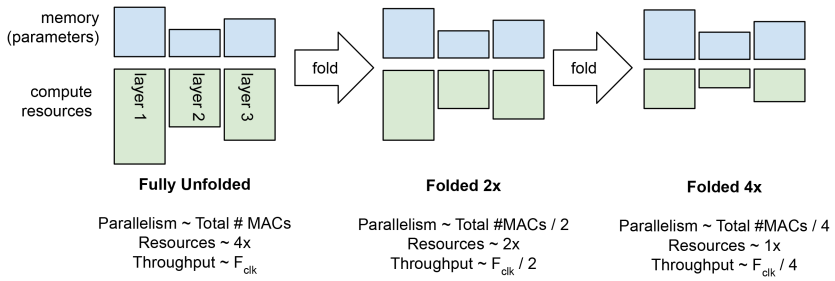


Fig. 3. Example of performance-resources trade-offs in a streaming dataflow architecture.

accelerator. MP is inherently lower latency, however from a software interface point of view, DP is more convenient as the interface to each accelerator is the same.

2.3 Dataflow Accelerator Performance Scaling via Folding

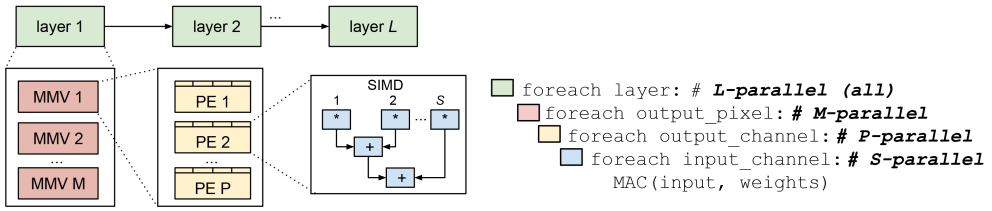
A key feature in the DF acceleration paradigm is the ability to generate architectures with different performance and resource characteristics to fit particular system-level requirements or resource budget constraints. In DF frameworks such as FINN [5, 58] this is achieved through time multiplexing, also called *folding* in FINN terminology. First, consider the case where every operation for a DNN is allocated to a dedicated compute unit, and the whole system is clocked at some clock frequency F_{clk}. Such an accelerator would be able to process one entire input sample per clock cycle (initiation interval II = 1) and yield a throughput of F_{clk} samples per second. Although this is possible for smaller DNNs, the resource cost is prohibitively expensive for larger models, and it is necessary to time-multiplex (or fold) the computation onto fewer resources by adding scheduling logic. For instance, time-multiplexing two operations onto a single compute unit will require half the hardware resources, but it will also yield half the throughput (II = 2). By controlling the degree of folding, we can obtain a multitude of designs with different performance-resources trade-offs, as illustrated in Figure 3. In practice, folding is achieved by controlling the degree of parallelism instantiated on reconfigurable hardware along different axes of the inference computation. This is illustrated in Figure 4 with a simplified view for DNNs with dense convolutions. Each layer has a set of parameters M, P, S that control the degree of parallelism along a particular axis, which must be chosen in a way such that: (1) a balanced streaming pipeline is obtained across layers, (2) the desired throughput is achieved, and (3) the total resource footprint fits within the given budget.

2.4 Design Flow for DF Accelerators targeting the Datacenter

Larger FPGAs targeting the datacenter tend to have a different construction than smaller ones, with implications on the performance of DFAs implemented on them, as we shall see in Section 3. We highlight some structural and CAD-related features of these systems here.

Figure 5 exemplifies a Xilinx DC-class FPGA constructed using a **stacked silicon interconnect (SSI)** [47] process whereby multiple FPGA dies, called **Super Logic Regions (SLRs)**, are mounted on a silicon interposer and connected through long wires across the interposer, denoted as **Super Long Lines (SLLs)** in the figure. Some SLRs can access DDR/HBM memory by implementing a controller in FPGA fabric.

Computational kernels are implemented for these FPGAs using computer-aided design software, such as Xilinx Vitis [29], which takes an accelerator as input, usually in C++ code but possibly in HDL (Verilog/VHDL) or in pre-synthesized form. We distinguish between two types of dataflow in Vitis designs: **embedded** and **explicit**. In **explicit dataflow** designs, the processing pipeline is



(a) Simplified hardware view. (b) Pseudocode view.

Fig. 4. Degrees of parallelism available for dense convolutions.

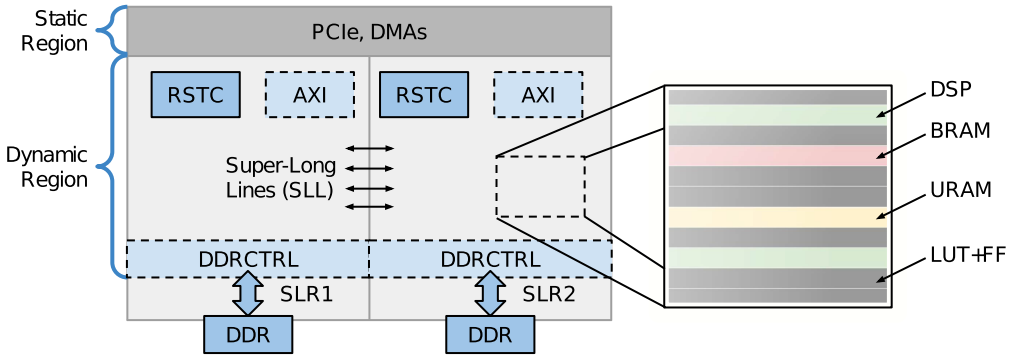


Fig. 5. Example DC FPGA platform using large, multi-SLR FPGA. SLRs are connected with long wires and consist of a columnar architecture of interleaved resource types. Vitis implements user kernels and utility logic in the dynamic region. The static region is called the Shell and is pre-configured.

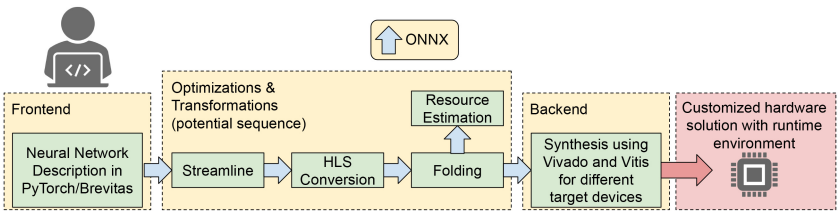


Fig. 6. FINN compiler flow.

assembled from multiple Vitis kernels connected together through AXI Streams. In designs with **embedded dataflow**, the logic itself is structured as a dataflow pipeline but the entire design is packaged into a single Vitis kernel. Embedded dataflow is specified using the HLS DATAFLOW pragma in C++ code.

To implement the accelerator in the dynamic region, Vitis makes any necessary connections for kernels to each-other, to DDR memory and to the host through the PCIe interface and DMAs implemented in the static (pre-configured) region. Kernels are fully synchronous and communicate through AXI interfaces. Vitis provides clocks and resets to kernels by implementing dedicated logic in each SLRs dynamic region.

2.5 The FINN Compiler

Xilinx provides an experimental open source framework to generate DF accelerators on FPGAs, which we refer to as the *FINN compiler* [33]. The FINN compiler has a highly modular structure as shown in Figure 6, which allows the user to interactively generate a specialized DF architecture

for a specific DNN. The framework provides a frontend, transformation and analysis passes, and multiple backends to explore the design space in terms of resource and throughput constraints. Brevitas [40], a PyTorch library for quantization-aware training, is the *frontend* used in this work. It enables training DNNs with weights and activations quantized down to a few bits, then exports the trained network into the **intermediate representation (IR)** used by the FINN compiler. The *transformation and analysis passes* help to generate an efficient representation of the DNN. Finally, the *backend* contains a code generator that creates synthesizable accelerator descriptions, which can be implemented as either a standalone Vivado IPI component or integrated into various shells, including Xilinx Alveo boards and PYNQ embedded platforms.

2.5.1 Frontend and Intermediate Representation. First the DNN model must be converted into the IR of the FINN compiler. The frontend stage takes care of this by converting the PyTorch description into the IR, called FINN-ONNX. This IR is based on ONNX [10], an open-source interchange format that uses a protobuf description to represent DNNs. It comes with several standard operators and allows the user to easily create their own operators to customize the model. The nodes represent layers and edges carry outputs from one layer to become inputs to another. The feature to customize the ONNX representation is used in the framework to add application specific nodes and attributes. Each node is tagged with the quantization of its inputs, parameters (weights and activations) and outputs to enable quantization-aware optimizations and the mapping to backend primitives optimized for quantized computation. During the compiler flow the nodes will be transformed into a backend-specific variants via a series of transformation passes.

2.5.2 Transformation and Analysis Passes. The main principle of the *FINN compiler* are graph transformation and analysis passes, which change or analyze the IR of the model. A pass is a function that takes the IR graph as input and either (a) *transforms* the DNN by looking for a certain pattern, changing the graph in a specific manner and outputs the modified graph, or (b) *analyzes* the DNN to produce metadata about its properties. To bring the model into a representation from which code can be produced and finally the hardware accelerator can be generated, various transformations must be applied. The main transformations involved are summarized below.

Streamlining. Although the PyTorch description of the network is mostly quantized, it may still contain some floating-point operations from, e.g., preprocessing, channelwise scaling, or batchnorm layers. To generate a hardware accelerator from the model, these floating-point operations must be absorbed into multi-level thresholds, so that a functionally identical network of integer operations is created. The transformation to achieve this is called *streamlining*, as described by Umuroglu and Jahre [59]. During streamlining, floating-point operations are moved next to each other, collapsed into a single operation and absorbed into succeeding multi-thresholding nodes.

Lowering and conversion to HLS layers. Next, high-level operations in the graph are *lowered* to simpler implementations that exist in the FINN hardware library. For instance, convolutions will be lowered to a sliding window node followed by a matrix-vector node, while pooling operations will be implemented by sliding window followed by an aggregation operator. The resulting graph now consists of layers that can be converted to hardware building block equivalents. Each node corresponds to a Vivado HLS C++ function call, from which an IP block per layer can be generated using Vivado. The resources utilized by each hardware building blocks can be controlled through specific attributes passed from FINN to Vivado. For example, multiplications can be performed with LUTs or DSP blocks, and parameters can be stored in distributed, Block, or Ultra RAM.

Folding and resource Estimation. The folding process (Section 2.3) assigns compute resources to each layer to obtain the desired throughput with a balanced pipeline. The underlying Vivado HLS library that provides the hardware building blocks for FINN supports controlling the degree

Table 1. Neural Networks Accelerated in This Work

Topology	Precision*	Model size (MB)	Accuracy (% Top-1)	GOps	FINN nodes
MobileNetV1 (MN)	W4A4	2.1	70.39	1.1	115
ResNet-50 (RN-50)	W1A2	11.25	67.27	6.8	277

*WxAy indicates quantization: x-bit weights, y-bit activations.

of parallelism along the P and S dimensions from Figure 4(b), called PE and SIMD, respectively, in FINN. To enable per-layer specialization without reconfiguration and minimize latency, FINN creates dedicated per-layer hardware interconnected with FIFO channels, thus the outermost loop across L layers is always fully pipelined.

Once the folding is specified, resource estimates can be produced for each node. There are several ways to estimate the resources. Even before IP blocks are generated from the HLS layers, an estimate of the resources per layer can be made by using analytical models based on the concepts from the FINN-R paper [5]. Estimations can also be extracted from Vivado HLS after IP generation, though these results are still estimations that may differ from the resource usage of the final implementation due to synthesis optimizations. Finally, FINN allows high-quality resource estimates to be obtained through out-of-context synthesis of nodes, at the expense of runtime. The most suitable estimation method depends on the context (e.g., prototype versus production phases).

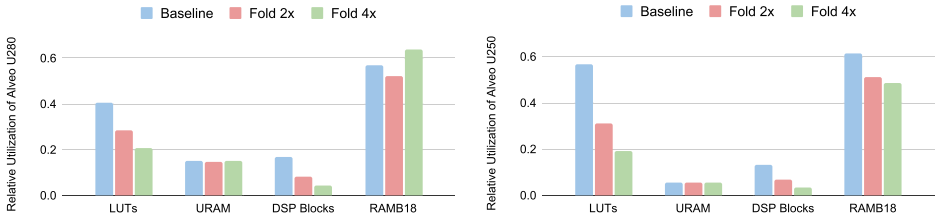
2.5.3 Backends. Backends are responsible for consuming the IR graph and backend-specific information to create a deployment package, also implemented using the transformation concept. To get the inference accelerator, between the layers FIFOs are inserted, which can be sized automatically by the FINN compiler. Afterwards the single IP blocks are stitched together and synthesized. The stitched IP can be manually integrated into a system, or inserted into an appropriate shell for the target platform. If the target platform is an Alveo card, then the design is exported as a Vivado **Design Checkpoint (DCP)**, followed by generation of Xilinx Vitis [29] object files and linking.

2.6 FINN-Accelerated ImageNet Classification

We utilize two deep neural networks for ImageNet classification as discussion vehicles to target DC accelerator cards throughout the remainder of this work: **MobileNetV1 (MN)** and **ResNet-50-v1.5 (RN-50)**. The key properties of these image classification DNNs are summarized in Table 1. We consider these DNNs relevant because of their widespread use in modern AI applications, where they are utilized both as classifiers and as backbones for more complex tasks like object detection. Both networks are part of the MLPerf DNN benchmark suite.

MN was first introduced by Howard et al. [22] in 2017 as a light weight DNN targeting the ImageNet [12] dataset. It has a repeated structure of **depthwise-separable (DWS)** convolution building blocks. Each DWS convolution block consists of a depthwise and a pointwise convolution followed by a batchnorm and ReLU block. The model also has a convolutional layer at the input and an average pooling followed by a fully connected layer at the output. In this work a reduced-precision implementation of MN from Brevitas [40] is used where the weights and activations are quantized to 4 bits, except for the first layer, which has a weight precision of 8 bits. The FINN-accelerated MN achieves a top-1 accuracy of 70.39% and is available as open source from Reference [32].

RN-50 [21] is a deep CNN designed for high accuracy on the ImageNet [12] benchmark. Unlike MN, RN-50 has a non-linear topology. We utilize the same quantization scheme described in Reference [43], resulting in a dataflow accelerator consisting of 112 layers, and a top-1 accuracy of 67.27% after FINN streamlining. Implementations of this network are available as open source [32, 42].



(a) MN Folding (b) RN-50 Folding

Fig. 7. Effect of folding on resource utilization for each DNN. The LUT and DSP utilization drops proportionally to the folding factor, while memory utilization is unchanged (MN) or slightly reduced (RN-50).

3 CHALLENGES IN SCALING UP DATAFLOW ARCHITECTURE PERFORMANCE

Ideally, a dataflow DNN inference accelerator should be able to utilize all the FPGA fabric available on the target device and simultaneously achieve high operating frequency, therefore maximizing performance per unit of cost. In this section, we highlight specific problems preventing DFA designers from achieving these requirements.

3.1 Limitations of Folding

There are practical limitations to folding DF-style accelerators that arise from the resource mix and granularity of a particular FPGA. Here, we use the FINN-accelerated MN and RN-50 (described in Section 2.6) to give an example. Figure 7 presents the resource utilization of each of the two DNNs in three different folding solutions: *Baseline*, *Fold 2×* (2× the II of Baseline), and *Fold 4×* (4× the II of Baseline). For each folding solution and DNN, we implement 4 resource allocation configurations (using DSPs or LUTs for multiplications, and using URAM or BRAM for sliding windows and FIFOs). The charts present the average resource utilization of these four configurations, and illustrates scalability limits in regards to resource usage and throughput. For MN, Baseline achieves II = 112 kcycles per frame, with the parallelism of the first layer limiting further performance increase, i.e., the first layer is fully unfolded. For RN-50, Baseline has II = 56 kcycles approximately. The limitation in this case is the number of DSP blocks required to implement the first convolutional layer. At this II, more than half of one SLRs DSP blocks are allocated to this particular layer, and increasing the performance further (by unfolding) eventually reaches the DSP resource availability on one SLR while other resource classes are underutilized. Splitting the layer across two SLRs to access more DSPs would lead to drastically reduced performance through $F_{\max}$ degradation (see Section 3.3). Because in a DFA all layers must fold to similar II, a limitation to the folding of one layer applies to the DFA globally resulting in low resource utilization of the FPGA. This limitation is an outcome of the resource mix and granularity of the FPGA, and applies to any DF inference framework, although the throughput at which these limitations become apparent depends on the granularity of folding. Coarser-grained folding (such as FINN) will experience the limitations at lower throughputs compared to finer-grained folding (such as fpgaConvNet [60]).

If the target FPGA is large enough, then performance can scale up through data parallelism by implementing multiple instances of the accelerator in a single FPGA. Otherwise, as is the case in our examples, one potential solution to utilize left-over space is to implement an additional folded-down accelerator instance, assuming that the resource utilization scales with performance and the folded-down DFA meets latency constraints. For each DNN, we evaluate two folded down variants that reduce performance by factors of 2 and 4, respectively. Figure 7(a) illustrates that compute resource requirements (LUT/DSP) scale with the FPS as expected, while memory

utilization (BRAM/URAM) remains relatively constant, as number and precision of stored parameters remains the same. this was expected as shown in Figure 3. Figure 7(b) presents a similar result for RN-50 with the only difference that memory utilization slightly decreases when folding as the weight buffers change shape and map better to the shape of BRAMs. Readers should refer to Reference [31] for a discussion of the effect of folding on buffer shapes and memory utilization.

This inelastic memory utilization limits the scalability of DFAs to smaller FPGAs. In both folded cases (Fold 2× and Fold 4×), the overall relative reduction in resource utilization is less than the relative reduction in FPS, which indicates that for streaming dataflow the optimal folding is the one with the highest FPS and lowest II, within topological and structural constraints. For this reason, in the remainder of this article, we utilize the baseline folding solutions where possible, and employ model parallelism to maximize resource utilization on any given FPGA target and enable accelerator portability to smaller FPGAs, as detailed in the following section.

3.2 Resource Balancing in the DF Pipeline

Resource *balancing* (as opposed to *allocation*, i.e., folding) refers to the process of selecting for each DF node a resource utilization profile, which can help optimize the cost of inference throughput in two ways. The first is through the maximization of accelerator throughput by route of operating frequency. Since the achievable frequency for any FPGA design is reduced by design congestion, it is advantageous to balance resource utilization between layers such that no class of resource is over-congested. Xilinx recommends a maximum utilization of 70, 50, and 80 percent of total available resources for LUTs, Flip-Flops and columnar structures (DSPs, Block, and Ultra RAM), respectively. In addition, the average of utilization of DSP, Block and Ultra RAM should not exceed 70%. In most cases, reducing utilization much below these limits does not have the opposite effect of increasing DFA frequency.

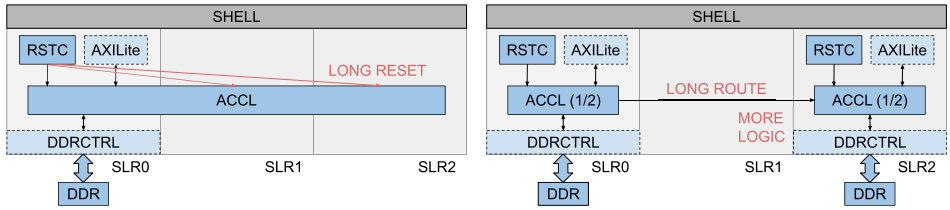
The second cost-of-throughput optimization route is through minimization of accelerator area. Careful selection of resource utilization per layer, coupled with placement constraints, enables a designer to "compress" the DF accelerator to its minimum FPGA area. This is beneficial either by creating sufficient space for multiple instances of an accelerator to co-locate the target FPGA, or, in future multi-tenant FPGAs by directly reducing the cost of deploying a single accelerator.

Resource balancing and partitioning should be performed simultaneously when mapping DFAs across multiple SLRs or FPGAs, since the partitioning decisions depend on the resource profiles of the layers. Unfortunately in the FPGA DNN inference and training literature, the two processes are typically decoupled, with balancing occurring as a fine-tuning phase. If the (achieved) objective of partitioning was to ensure congestion guidelines are respected, then subsequent resource balancing achieves very little frequency increase, and since partitioning is already set, no useful area decrease.

3.3 Timing Closure Challenges in SSI FPGA Design

The practical implementation of DFAs such as the FINN-style MobileNetV1 and ResNet-50 on DC-class FPGAs raises specific problems not encountered in the domain of embedded FPGAs. These problems arise due to a combination of HLS design entry, the large size of DC-class DFAs, and the non-monolithic structure of the SSI FPGAs themselves. In this context, often the DFA does not fit into a single SLR, and it is currently impossible for the high-level synthesis tool, e.g., Vivado HLS, to properly estimate where the die crossings will occur. Crossing from one SLR to another over a SLL incurs a relatively large propagation delay (compared to on-SLR routes), which can have an impact on operating frequency unless carefully considered in the design entry.

Furthermore, utility logic external to the kernel may also limit performance. Illustrated in Figure 8(a) is a situation where the reset signal limits the achievable frequency. Since the kernel



(a) Timing failure due to Resets

(b) Timing failure and extra logic due to naive placement of accelerator partitions

Fig. 8. Potential pitfalls in the implementation of large dataflow accelerators on multi-SLR FPGAs.

is fully synchronous it must be reset from a single source. If a kernel spans multiple SLRs, then so must the reset signal, whereby the magnitude of the problem increases with number of SLRs. This problem applies to multi-SLR DFAs where the dataflow is embedded, i.e., described in HLS C++ using the HLS DATAFLOW pragma, or as is the case with FINN, in a Vivado IP Integrator pipeline that is exported to a single Vitis kernel. For these cases, reset pipelining inside the kernel is a potential solution but it cannot currently be implemented automatically by the platform linking tool (Vitis or equivalent), because it does not have access to the kernel internal logic.

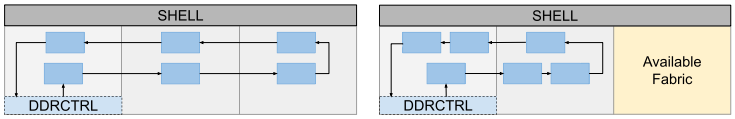
4 ELASTIC DATAFLOW ACCELERATION

We propose that the common requirement to solving all of the problems identified in the previous section is a transition in DFA design philosophy from monolithic kernels with embedded dataflow to explicit and fine-grained dataflow, whereby a DFA consists of multiple kernels that communicate through AXI Streams, such that no kernel spans more than one SLR. However, within the context of fine-grained explicit dataflow, it becomes challenging even for expert FPGA designers to find a good partitioning of DFA logic between the kernels, and the corresponding association of kernels to SLRs (and FPGAs), while simultaneously tuning the resource utilization profile of each kernel to minimize DFA area. We present Elastic-DF, a combination of automatic partitioning, automatic resource balancing, and a light-weight FPGA-to-FPGA communication fabric, which can efficiently map a DFA to a target single- or multi-FPGA platform, optimizing inference throughput per cost with minimal human intervention. Elastic-DF is designed to optimize the implementation of an already folded DFA, therefore the user is responsible for performing folding to their desired latency, and may provide multiple potential implementations for each node in the DNN graph, e.g., implementations using LUTs or DSPs for multiplication, or Distributed versus Block RAM for storage, which Elastic-DF can utilize in the process of resource balancing.

4.1 Partitioning Scenarios

We analyze partitioning scenarios to identify requirements for the Elastic-DF partitioner.

Single DFA on Single FPGA: Acceleration and Compression. When deploying a single DFA to a multi-SLR FPGA, the role of the partitioner is to segment the DFA into kernels and assign these to SLRs, which is non-trivial. Figure 8(b) illustrates a situation where, despite having overcome the reset bottleneck in Figure 8(a) by splitting the DFA into two kernels, naive placement of the kernels means the wires connecting them have to cross two SLLs and the width of an SLR, creating very long route delays. Suboptimal partitioning means both kernels require access to DDR memory so an additional controller must be instantiated at the cost of extra power dissipation and logic utilization.



(a) Congestion Equalization (b) Accelerator Compression

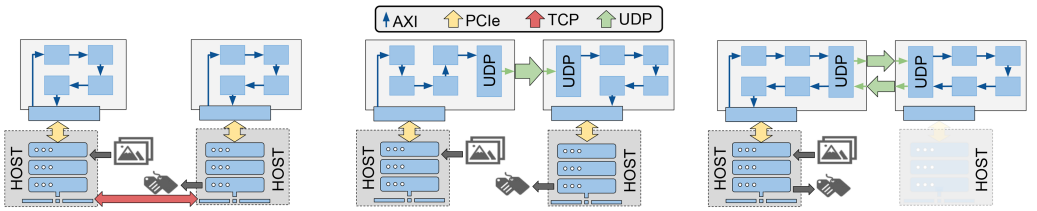
Fig. 9. Benefits of automatic partitioning of DF accelerators for single-node implementation. Congestion is always balanced between SLRs to maximize operating frequency. Space savings through accelerator compression can result in cost savings in a multi-tenant environment.

A better partitioning is illustrated in Figure 9(a), from which we derive the following requirements for the Elastic-DF partitioner. The size of the kernels and their allocation to SLRs must be chosen to equalize the congestion of the SLRs, and kernels must be placed such that no AXI Stream crosses more than one SLL, maximizing operating frequency. It may be advantageous to avoid instantiating a second DDR controller and the associated power and area penalty by co-locating all DF kernels that require DDR access in the same SLR—the partitioner should provide an option to do so. Also, since not all SLRs have access to external memory, the partitioner needs to allow for absolute placement restrictions for specified kernels.

Figure 9(b) illustrates how resource balancing can create free space in the form of an empty SLR after the DF kernels have been re-allocated to SLRs 1 and 2, maintaining a balanced congestion. While multi-tenancy is not yet a feature of DC FPGAs, it is reasonable to assume that the SLR is a natural granularity for multi-tenancy to exploit, and therefore accelerator compression creates a cost reduction opportunity by minimizing the number of SLRs needed for a given DFA. To support compression, the Elastic-DF partitioner must allow multiple possible implementations for each node in the DF graph, from which exactly one can be instantiated in the final design.

Multi-DFA on Single FPGA. The same optimization criteria apply when implementing multiple DFAs in a single FPGA. This scenario requires partitioner support for disjoint subgraphs, whereby each of the DFAs is a subgraph and the entire graph is partitioned together to ensure the best quality of results. As a side-effect, accelerators consisting of multiple dataflow sections communicating through external memory could be partitioned as well.

Multi-FPGA. When a (group of) DFAs exceed the resources of a single FPGA, automatic partitioning should enable their separation into parts executing on different FPGAs. In this case, the partitioning algorithm must also take into account the maximum communication throughput between the FPGAs, in addition to the congestion minimization criteria. Figure 10 exemplifies three possible methods of implementing model parallelism for one DFA deployed to two FPGAs: **Software Model-Parallel (SWMP)**, **Hardware Model-Parallel (HWMP)**, and **Transparent Model Parallel (TMP)**. In Figure 10(a) the DFA is partitioned between two FPGAs without direct communication. Inputs are ingested on one node, intermediate results are produced and transported over the host network to the second node, where they are loaded to the FPGA and results are produced. In Figure 10(b), we implement direct FPGA to FPGA Ethernet connectivity, removing the host involvement for intermediate result transport. Finally, Figure 10(c) illustrates transparent model parallelism, whereby the DFA is partitioned into three consecutive segments: the first and last execute on one FPGAs, while the second segment executes on the other FPGA. Intermediate results between segments are transported using the FPGA-to-FPGA Ethernet. In this way, results are delivered to the same host that provides the inputs, and the software interface to the accelerator is identical to a single-FPGA accelerator.



(a) SWMP (b) HWMP (c) TMP

Fig. 10. Model parallel variations implementable with dataflow inference accelerators between two FPGA-equipped computing nodes. Software Model-Parallel (SWMP) utilizes the host NIC for activation transport, while Hardware Model-Parallel (HWMP) and Transparent Model Parallel (TMP) utilize the FPGA for transport, which adds logic overhead. Note that TMP does not require host involvement on one of the nodes and has the same software interface as a single-node accelerator.

Each of the three approaches has advantages and disadvantages. SwMP has less FPGA design complexity (no Ethernet) but moves more data to and from the host. In TMP, the host on the second node has no involvement in computation, and only needs to perform configuration of the FPGA logic and network parameters, after which it can be switched to a low-power state or allocated to other DC users that do not require FPGA access. This facilitates resource disaggregation in the datacenter. However, the less constrained nature of HwMP may create an operating frequency advantage. Lacking a theoretical framework to differentiate between these three approaches to model parallelism, the Elastic-DF partitioner must support all three approaches, and we will derive guidelines from empirical evaluation in subsequent sections. TMP can be enforced through co-location constraints as described in previous scenarios.

4.2 Elastic DF Partitioner

Graph partitioning algorithms have been studied extensively in the context of many application domains. Due to its NP-hard nature, the problem size (number of nodes in graph, number of partitions) is a key factor in determining the range of applicable algorithms for partitioning. Relatively small problems may be solved by exact methods like enumeration, whereas larger problems require heuristic methods, such as iterative greedy refinement and simulated annealing. When problem scale permits, exact methods are preferable to heuristics, because they can determine the optimality of a solution or the (in)feasibility of the problem.

The usual approach among DNN frameworks targeting multi-FPGA execution [34, 35, 54, 62] is to utilize greedy, dynamic programming or otherwise heuristic solvers (see Table 8 in Section 7), which enables them to map a single DNN through model parallelism to an arbitrarily large number of FPGAs, extracting the maximum throughput and minimum latency from the single DNN.

Since our graph partitioning problem does not (generally) exhibit optimal substructure [11], such heuristic solvers are unlikely to identify an optimal solution especially if the graph is coarse, i.e., consists of DNN layers instead of individual multiply-accumulates. Conversely, we observe that our partitioning problem, including all the requirements previously identified, can be expressed as an ILP and solved exactly, which is desirable, at the expense of potentially very long solver runtime when a large number of DFAs need to be mapped to a large number of FPGAs. To constrain the size of the problem as seen by the ILP solver, we divide the problem into two steps. We first map a relatively small number of DFAs to a relatively small number of FPGAs to create a highly efficient inference "tile." This tile can then be replicated through data parallelism up to arbitrary scales. The problem then becomes to discover the parameters of the tile: its size (number

of DFAs, number of FPGAs), the partitioning of each DFA, and the placement of the kernels resulting from partitioning onto the tile FPGAs and SLRs therein. The next sections describe how we partition for a tile of known size, and how we discover tile size.

4.2.1 Partitioning DFAs to a Tile. We define the partitioning problem through linear constraints and an objective function for the optimization. Given a graph where nodes are DNN tasks (e.g., layers), the mapping of graph nodes to SLRs can be expressed as binary decisions indicating whether a particular version of a task node is present or not in a particular compute node. The fact that each task node must be instantiated once and only once is easily stated as a linear equality. Resource constraints are expressed as inequalities forcing maximum SLR utilization below Xilinx-recommended limits (but we do allow user overrides). Similarly, we express communication resource limits as linear inequations, where the constraints are expressed in Gbps when the path is off-chip, or in number of SLLs when the path is on-chip. Also, by using linear constraints on specific nodes, we are able to express both absolute placement and relative (co-location) anchors.

The optimization objective of our ILP formulation aims to minimize the sum of the costs of graph cuts, which can be set differently depending on whether the communication is on- or off-chip. These costs reflect the preference of using one or another physical connection and enable the implementation of high-level partitioning strategies. For example, within each FPGA chip, a high cost could be assigned to SLR crossings and, as a consequence, the number of tasks edges mapped to these crossings would be minimized. In multi-FPGA systems, we may also seek to minimize the number of FPGAs required for the accelerator, which can be achieved by setting connection costs between cards greater than the total cost of placing all the tasks in one device.

By design, solving the ILP as formulated here does not minimize but only restricts the required communication between nodes to the physically available values (e.g., number of SLLs or Ethernet bandwidth). Similarly, the ILP solver does not directly minimize the number of SLRs in use, but this can be achieved by co-locating the start and end nodes of the graph, which makes communication cost minimization equivalent to minimizing the number of SLRs.

The complete mathematical ILP formulation can be found in Appendix A.

4.2.2 Optimizing Tile Size. The core partitioner as presented so far utilizes a fixed description of the (multi-)FPGA tile as an input. However, as discussed in Section 3.1, mismatches between the tile available resources and DFA required resources may mean the platform is being under-utilized. We therefore embed the ILP partitioning in a tile-size optimization algorithm, which maximizes the utilization of FPGA fabric in the tile, and therefore the performance per cost metric important in datacenters. The tile hardware is defined by: the type of FPGA, the connectivity pattern between FPGAs, e.g., daisy-chain, and the maximum scale (number of FPGAs). Algorithm 1 describes an iterative approach to searching for the optimum scale, where the number of DFA instances and scale are increased until the efficiency passes some user-set threshold or the maximum scale is reached. We take advantage of the fact that exact ILP solvers can rapidly identify unfeasible combinations of platform and DFA(s) and terminate partitioning early.

4.2.3 Partitioner Implementation. The developed ILP formulation is generic, i.e., can partition any DFA for any platform as long as these are specified appropriately. For this work, the partitioner was implemented in Python using the ILP solver provided by the mip [57] module. The resulting partitioner is relatively fast. For a graph consisting of 100 nodes targeting a platform of 10 SLRs with resource utilization approaching the target resource limits, the runtime is only a few seconds. The partitioner can be easily integrated into any DNN compiler framework that provides per-layer resource estimation and infrastructure to pass placement constraints into the design at build time.

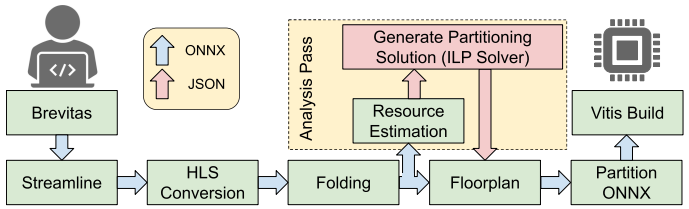


Fig. 11. Integration of ILP partitioner into FINN ecosystem.

ALGORITHM 1: Tile Sizing

Result: tile size: number N_{FPGA} of FPGAs; number N_{DFA} of DFA instances
define the platform P with maximum scale N_{FPGA}^{max} and DFA model M;
define minimum acceptable utilization U_{min};
N_{DFA} = 1; N_{FPGA} = 1; U_{best} = 0;
do
  s = Partition(M, P, N_{FPGA}, N_{DFA}) ; // partition N_{DFA} instances on N_{FPGA} FPGAs
  if *s exists* **then**
    U = GetUtilization(s); // calculate how much of the FPGAs are utilized
    N_{DFA} += 1; // increase the number of instances (speculatively)
  else
    N_{FPGA} += 1; // if partitioning failed, increase the number of FPGAs
  end
while $U < U_{min}$ *and* $N_{FPGA} <= N_{FPGA}^{max}$;

4.2.4 FINN Integration. Figure 11 illustrates how our partitioner integrates with the DNN compiler FINN. We implemented a custom FINN analysis pass that instantiates the ILP partitioner and translates between the FINN ONNX DFA representation and the partitioner DFA representation. The partitioner task graph is produced by post-processing the JSON generated by the FINN resource estimation from the input ONNX, while the task connectivity information is extracted from ONNX directly. The compute graph is generated by custom code that describes multi-node Alveo systems, and anchors are optionally provided by the user as arguments to our analysis pass. The partitioner associates each FINN layer with a specific FPGA and SLR on the platform, and this information is encoded in a FINN-specific JSON and applied as custom attributes to the ONNX model using the existing FINN Floorplan transform. Later in the DFA build process, these attributes instruct FINN to separate the ONNX model into partitions, i.e., contiguous sections consisting of layers that reside on the same SLR. Subsequently each partition is converted into a Vitis kernel and linked by the FINN backend to produce a bitstream.

4.3 VNx: A FPGA-to-FPGA Communication Infrastructure

To enable TMP and HwMP in our designs, a direct communication between the nodes is desirable. Xilinx FPGA Alveo are equipped with two QSFP28 interfaces directly connected to the FPGA and enable Ethernet peer-to-peer communication with other devices. In this section, we describe the Vitis network layer [45], which implements transport of intermediate results for our MP designs over the UDP/IP protocol. We chose UDP because of its relatively reduced complexity, which enables a compact implementation in an FPGA. We provide VNx to the FPGA community as open-source code that other researchers can utilize to implement UDP network connectivity between dynamic regions of Alveo platforms. Figure 12 depicts the structure of VNx:

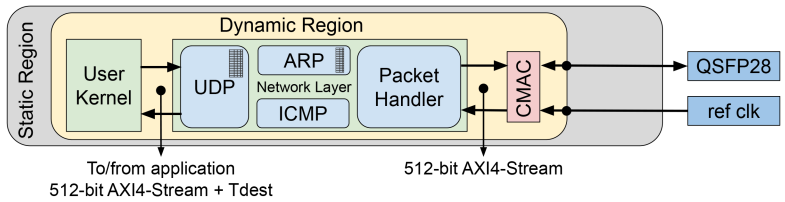


Fig. 12. Vitis user Kernel Network block diagram.

Gigabit Transceivers. VNx requires access to **Gigabit Transceiver (GT)** pins from the dynamic region. Currently the Alveo U50, U250, and U280 shells support this functionality.

Media Access Control (CMAC) Kernel. This kernel provides link bring up logic, 100 Gb/s Ethernet MAC, and translates the physical signals from a GT pins to a 512-bit AXI4-Stream and vice versa. Configuration and statistics registers are accessible via an AXI4-Lite interface.

The Network Layer Kernel. This kernel (middle of Figure 12) interfaces directly with the CMAC and is composed of several submodules. The Packet Handler parses, forwards, or drops incoming packets as appropriate. For outgoing IP packets, the MAC address is resolved from the IP address by querying the **Address Resolution Protocol (ARP)** submodule, which contains a 256-entry table that is accessible from the host through AXI4-Lite. If a valid MAC address is found, then the Ethernet header is populated and the packet is forwarded to the CMAC, otherwise is discarded, and an ARP request packet is generated. Finally, the UDP module associates UDP IDs to user kernels using a user pre-populated 16-entry connection table that associates an ID with a triplet (destination IP, source port, destination port). Incoming packets from the network are parsed and the connection table is queried to obtain the associated ID. If the ID is found, then the payload is forwarded to the application and the sideband channel *dest* in the AXI4-Stream indicates the ID of the connection, for correct switching of the payload to the appropriate compute unit. In the transmitting direction, the opposite happens. When the payload is received by the UDP module, the table is queried using the *dest* field. If the entry is valid, then a UDP packet is generated with the information stored in the table. Otherwise the payload is dropped.

UDP was chosen because of its small footprint due to its stateless and light-weight nature. Table 2 provides a breakdown of the VNx resource consumption, which is a small fraction of the resources available on a SLR of any Xilinx accelerator cards. The same table lists for comparison a full-fledged TCP/IP stack [18], which provides reliable packet transport however in exchange for higher latency, significantly higher resource consumption (4×), plus external memory interface, which creates additional placement constraints and with that timing closure challenges as previously discussed.

With regard to throughput, our benchmarking experiments indicate that the throughput at application level reaches 90% of the Ethernet theoretical maximum of 100 Gb/s (and 100% of application-level maximum) with payload size bigger than 448 bytes and reaches 97 Gb/s for packets larger than 1 Kbyte. With regards to latency, from the application point of view, a point to point connection is 0.526 μs. However, when going through a 100 Gb/s switch the latency is as high as 1.376 μs. For DC dataflow inference acceleration, where typical latency is in the millisecond range, the additional latency caused by VNx transport is negligible. The power dissipation of VNx at full throughput is 6 W.

Finally, the IEEE Std 802.3BJ-2014 [51] establishes a peak **bit error ratio (BER)** of 10^{-12} for 100 Gbps Ethernet, which corresponds—given a simple yet realistic error model (link speed × BER)—to one packet lost every 10 s on average. We carried out a continuous transmission for over 15 h

Table 2. Breakdown and Comparison of Resource Utilization between VNx (UDP) and ETH TCP Stack

Kernel	VNx UDP Stack (ours)			TCP Stack [18]			
	LUT	FF	BRAM	LUT	FF	BRAM	URAM
CMAC Kernel	12,038	37,449	68	13,538	53,891	34	9
Network Kernel	23,181	48,820	115	110,934	176,191	813	1
Packet Handler	4,273	13,649	68	19,741	45,106	200	0
ARP	1,923	4,257	7	614	1,239	3	0
ICMP	2,646	5,617	0	1,496	1,350	8	0
Transport Layer	7,981	12,729	40	47,334	70,879	480	0
Total Utilization	35,219	86,269	183	124,472	230,082	847	10

between FPGAs with no packet lost from almost 700B packets, which indicates real-world BER is much lower than the worst-case estimate.

5 EXPERIMENTAL SETUP AND EVALUATION METHODOLOGY

The performance of distributed inference depends on multiple layers of the software-hardware stack and is therefore difficult to accurately simulate or model. We therefore focus on practical implementation of distributed DNN inference accelerators and their evaluation.

5.1 ETH Zurich XACC

We utilize the **Xilinx Adaptive Compute Clusters (XACC)** [44] as a platform for evaluating our single- and multi-node accelerators. XACC is a research initiative that provide critical infrastructure and funding to support novel research in adaptive compute acceleration for **High-performance Computing (HPC)** at four prestigious universities worldwide. The XACCs are composed of high-end servers, Xilinx Alveo accelerator cards and high-speed networking, the configuration of each of them is tailored to enable research around a particular area.

In this work, we utilize the ETH Zurich XACC, which is composed of four FPGA-equipped servers, each of which has a mix of Alveo U250 and Alveo U280 cards, ten in total—see Table 3 for details. Each Alveo card features two 100 Gb/s Ethernet interfaces: one is connected to a Cisco Nexus 9336C-FX2 switch and the another is connected to its Alveo neighbor. Every server also has a 100 Gb/s **Network Interface Card (NIC)** connected to the switch. This configuration allows users to explore arbitrary network topologies for distributed computing. Our work makes extensive use of the FPGA-to-FPGA connectivity provided by XACC, and we utilize both U250 and U280 Alveo cards to demonstrate the capabilities of the Elastic-DF flow. The software environment is based around Ubuntu 18.04 and includes many software frameworks for FPGA accelerator deployment: XRT, PYNQ for Alveo, Jupyter Lab. Dask, and InAccel Coral [28] provide scale-out capabilities.

5.2 FPGA Execution Orchestration with InAccel Coral

Coral is a fast and general-purpose accelerator orchestrator. It provides high-level APIs in C/C++, Java and Python, which enable a user (the client) to make acceleration queries to a unified execution engine (the server) that supports every heterogeneous multi-accelerator platform. InAccel also provides a runtime specification that vendors can use to advertise system hardware resources to Coral. It aims to specify the configuration, and execution interface for the efficient

Table 3. ETH Zurich XACC Infrastructure

Node name(s)	CPU	Frequency	Cores	RAM	Alveo cards
alveo0	2× Intel Xeon Gold 6248	2.50 GHz	40	376 GiB	N/A
alveo1, alveo2	2× Intel Xeon Gold 6234	3.30 GHz	16	376 GiB	2 × U250
alveo3, alveo4	4× Intel Xeon Gold 6234	3.30 GHz	32	376 GiB	1 × U250 + 2 × U280

management of any accelerator-based hardware resource (e.g., FPGAs), without customizing the code for Coral itself. In Coral, client applications call an accelerator on a local FPGA as if it were a software function, making it easier for the user to accelerate distributed applications and services. Coral users define a platform, i.e., the accelerators that can be called remotely with their arguments and configuration parameters. Coral clients and servers can run and communicate with each other in a variety of environments—from bare-metal Linux servers to containers inside a Kubernetes cluster—and applications can be written in any of Coral API's supported languages.

The key concepts of the InAccel runtime specification are the following: *resource*, *memory*, *buffer* and *compute unit*. InAccel offers a default OpenCL-based Xilinx FPGA runtime implementation, where each resource object represents a single FPGA device with a list of memories and compute units, respectively, entities that is dynamically updated upon reconfiguration of the FPGA. Last, *buffer* objects belong to a specific *memory* inside the *resource* context, which corresponds to a physical memory region on the FPGA board's DDR memories.

Dual-FPGA abstraction for Coral. To support multi-FPGA accelerators, we developed a custom InAccel runtime, enabling initialization-time configuration of the VNx and run-time synchronization between accelerator segments where applicable (HwMP, SwMP). In this new abstraction, the resource object hosts the context of two devices (e.g., 2 Alveo U280), instead of one, but also a unified memory topology is announced to Coral. Each bitstream artifact is now a tarball that includes two named binaries, while the kernel metadata contain directives that indicate in which binary they reside. By abstracting away all the lower level details, our Dual-FPGA Coral runtime is able to transform any dual-FPGA cluster to a single pool of accelerators. This enables us to run existing InAccel MLPerf test harnesses against our dual-FPGA accelerators with no changes to application code. This runtime can be extended in principle to any number of FPGAs.

5.3 Methodology

We evaluate several neural network accelerator implementations of the FINN-generated DF accelerators for the quantized MN and RN-50 classifiers.

We first evaluate single-accelerator multi-FPGA inference. The research objectives are: to validate the floorplanner's ability to model inter-FPGA connectivity when partitioning an accelerator between FPGAs, to compare direct FPGA-to-FPGA communication with through-host communication, and finally to compare the best-case throughput and latency of multi-FPGA model-parallel inference with data-parallel inference.

Next, we evaluate accelerator compression through resource balancing in the partitioner, targeting single-FPGA inference. We equate size reduction with a potential reduction in application cost. Whenever possible, we utilize the freed-up space to increase the number of accelerator instances implemented on the target system and report the overall increase in performance.

Finally, we evaluate the achievable operating frequency increases from removing the reset bottleneck through accelerator partitioning. We target single-FPGA inference on SSIT FPGAs and partition the accelerator across the available SLRs within congestion limits.

Table 4. Implemented ResNet-50 and MobileNetV1 Inference on Two Alveo U280s at XACC

Accelerator	Parallelism	CUs	$F_{\max}$	Peak FPS	Kernel Latency (ms)	End-to-End Latency (ms)	Average Power (W)	Est. CPU Power (W)
RN-50	DP (Fold 2×)	2	215	3513	2.72	5.32	134	
	TMP	1	205/220	3374	1.52	3.75	120	20.0
	HW-MP	1	220/235	3618	1.55	3.59	124	22.2
	SW-MP	1	215/230	3559	2.01	5.70	115	26.5
MN	DP	2	240	4195	2.11	4.42	100	
	TMP	3	225/230	5923	2.19	4.60	128	31.8
	HW-MP	3	220/220	5755	2.43	4.54	127	32.2
	SW-MP	3	230/225	5913	2.83	6.95	120	32.3

We model multi-FPGA platforms for partitioning as follows: each FPGA F_i in a system has at most two Ethernet connections to F_{i+1} and F_{i-1}, creating an Ethernet daisy-chain. The Ethernet interface is placed in the SLR where the GTH Transceivers reside, i.e., SLR2 of U280, or SLR1 of U50. Resource availability for each Alveo platform is derived from inspecting Vivado-generated design checkpoints. We find that true available resource counts are slightly higher than those reported in Xilinx documentation (UG1120). We apply absolute constraints to ensure no DMAs are placed in SLRs that do not have access to either HBM or DDR. Whenever both HBM and DDR are available, we assume use of HBM.

Resource estimates for DFA layers, required for partitioning, were gathered using the out-of-context synthesis estimation flow in FINN. We also experimented with HLS and analytic estimates and found that, in general, HLS-based estimates are better for RAM utilization, while the FINN model is more accurate for LUTs. Both are less accurate than out-of-context synthesis, as expected.

FPGA builds are performed utilizing a very high optimization effort level in Vitis 2020.1. In addition to setting Vitis to its highest optimization level (−O3), we further increase implementation effort with specialized settings: we activate the PHYS_OPT_DESIGN phase, activate ExploreWithRemap directive for OPT_DESIGN phase, activate Explore directive for PLACE_DESIGN, ROUTE_DESIGN, and PHYS_OPT_DESIGN, activate post-route TNS cleanup. Kernels are explicitly assigned to SLRs according to the partitioning solution using a TCL recipe. Each build is run multiple times with target frequencies ranging from 180 to 250 MHz in increments of 10 MHz, and we keep only the build with highest achieved F_{max}. We perform this target frequency sweep to eliminate the typically high variability in the resulting F_{max} when changing the Vitis target frequency. For U250 and U280 builds, we utilize the U250_XDMA_201830_2 and U280_XDMA_201920_3 platforms, respectively.

6 EVALUATION RESULTS

6.1 Model Parallel Acceleration

We apply our partitioning flow to ResNet-50 and MobileNetV1 targeting two Alveo U280 FPGAs with peer-to-peer 100 Gbps Ethernet connectivity. The resulting multi-FPGA inference designs were implemented and their performance is illustrated in Table 4. In the DP evaluation, the default Inaccel Coral runtime is utilized to distribute the data to the FPGAs and gather results. TMP and HwMP utilize VNx for direct FPGA-to-FPGA communication. The Inaccel Coral multi-FPGA runtime handles data movement via PCIe and host memory for SwMP. We note that TMP requires two-way communication, while SwMP and HwMP send traffic in a single direction.

The baseline data-parallel ResNet-50 implementation for the U280 system utilizes a folding to II = 112 kcycles per frame (denoted Fold 2× in Section 3.1), which allows the accelerator to fit in a U280. With data-parallel execution, the aggregate throughput of the two U280 FPGAs is slightly above 3.5 kFPS. We observe linear increase of performance between one- and two-FPGA systems.

The ResNet-50 model-parallel accelerators (TMP, HwMP, SwMP) are folded to II = 56 kcycles, and do not fit a single U280 FPGA, and are therefore automatically partitioned across two devices. Approximately 40 Gbps are required for the communication between the FPGAs in our partitioning solution. We observe that the best throughput achieved through model parallelism (using HwMP in this case) is slightly higher than DP achieves, while kernel latency is approximately halved. This is in line with the expectations—the mechanism by which performance is improved in this case is by enabling the implementation of a smaller number of accelerator instances (1 instead of 2) but with higher-performance folding, which optimizes latency but not throughput. The slight increase in FPS is due to higher operating frequency caused by reduced OCM congestion in the model-parallel cases, which implement a single CU therefore only store the DNN weights once, whereas each of the 2 CUs of the data-parallel implementation stores its own copy of the DNN weights.

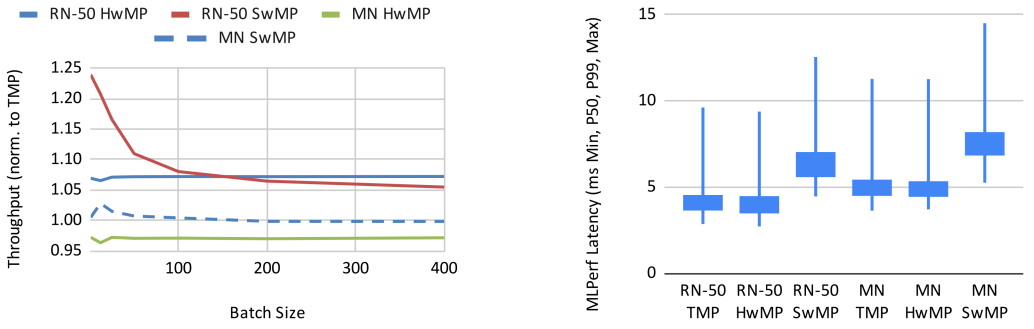
The baseline data-parallel MobileNetV1 implementation for the U280 system utilizes a folding to II = 112 kcycles per frame - the lowest achievable II given the available parallelism mechanisms in FINN. At this folding, the accelerator comfortably fits the U280, utilizing approximately 60% of fabric resources. The aggregate throughput of the two U280 FPGAs is slightly below 4.2 kFPS. For MobileNetV1, because the accelerator cannot be further unfolded, we perform automatic partitioning to maximize density (and therefore throughput) at the current folding, and find that we can fit three instances of the accelerator in two U280s, one of which is split across the two FPGAs. We measure a 41% increase in throughput (with TMP) compared to data-parallel execution, while latency is only slightly decreased, proportionally with the difference in operating frequency between the DP and MP systems.

6.2 Comparing TMP, HwMP, and SwMP

We compare the three model-parallel approaches to derive guidelines for their utilization in practice. We first observe that the practical operating frequency for the TMP and HwMP dual-FPGA systems is the minimum of the two FPGAs $F_{\max}$. Because UDP does not provide flow control, we find that using unbalanced frequencies on the two sides of VNx links leads to either packet loss in the Ethernet link or buffer overflows in the FPGA dataflow. However, when both sides operate at the same frequency we find the system is robust. SwMP can operate its FPGAs at different frequencies without issue.

Despite the slightly higher operating frequencies used, Table 4 indicates that SwMP dissipates less board power than either HwMP or TMP. The difference accounts for the two VNx kernels on either side of the peer-to-peer link, which dissipate approximately 12 W together. The observed difference is smaller than this value, because there is more power dissipated in data movement activity between FPGA and host in SwMP compared to TMP/HwMP. While the XACC does not provide CPU power measurement infrastructure, we measured CPU utilization for TMP, HwMP, and SwMP at batch 400, which we applied as a scaling factor to the 260-W TDP of the XACC server processors (2× Intel Xeon Gold 6234) to derive the CPU power estimates in the table. We observe that TMP dissipates the least, and SwMP the most CPU power. The effect is less pronounced for the MobileNetV1 system, where two of the three accelerators are not split across FPGAs, therefore their control workload does not change between HwMP/TMP and SwMP.

Figure 13 presents a more detailed analysis of throughput and latency. Figure 13(a) illustrates throughput of HwMP and SwMP normalized to TMP for both MobileNetV1 and ResNet-50. For both accelerators, but more pronounced for ResNet-50, we observe SwMP outperforms for small



(a) Throughput Comparison

(b) Latency Comparison

Fig. 13. Comparison of SwMP, HwMP, and TMP for RN-50 and MP on two Alveo U280s at XACC, using Inaccel Coral. Latency and throughput are measured using MLPerf Single Stream and Offline benchmarks, respectively.

batch sizes. At larger batch sizes the differences between the MP variants stabilize to approximately the difference in their operating frequencies. With regard to latency, Figure 13(b) suggests that SwMP has much higher latency at the application level than TMP and HwMP, which perform equally well. The difference is due to the extra latency of data movement through the host. We also observe higher variability of latency, with the difference between 50th percentile and 99th percentile latency being approximately double for SwMP compared to the other MP variants. As a direct consequence of this, additional evaluation with the MLPerf Multi-Stream benchmark, which focuses on latency-constrained throughput, indicates that the ResNet-50 SwMP achieves approximately 40% fewer FPS when the 99th percentile latency is constrained to 50 ms, and approximately four times fewer FPS when constrained to 7 ms.

6.3 Effect of Partitioning on Logic Density

ILP versus Greedy. We evaluate how the Elastic-DF partitioner compares against a simple greedy partitioner as utilized in the distributed inference framework AIGean [36, 55] with regard to achievable logic density, without communication constraints. This set-up is indicative, given the focus on placement only, of monotone partitioning strategies, i.e., if layer i and k are placed onto a specific SLR, then any layer j is placed on the same SLR, where $i \leq j \leq k$.

The greedy partitioner packs each SLR with consecutive DFA layers until the Xilinx-recommended congestion limits are reached, then moves onto the next SLR. We perform this experiment using MN for all Alveo cards. For U50, the greedy partitioner was not able to map the DFA to the device, running out of Block RAM after placing 29 of the 31 layers in the MN DFA. For all remaining cards (U200, U280, U250) the greedy partitioner was able to fit the DFA into three SLRs. In contrast, our ILP partitioner was able to fit the DSA into all of the cards using 2 SLRs in each case, benefiting from the ability to generate a non-monotone partition, i.e., move the pipeline back and forth between the SLRs to optimize utilization.

Additional Compression via Resource Balancing. For this experiment, we enable the resource balancing feature of our ILP partitioner and evaluate its effect on the total size of FPGA required for implementing a DFA. We compare against our ILP partitioner with resource balancing turned off. Figure 14 presents a mix of estimated and measured performance for classification with MN with each of the four Alveo accelerator boards as target. For each board, we present the achievable FPS with and without resource balancing. For U50 and U200, the top of each bar indicates the

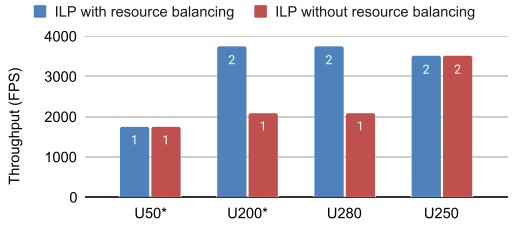


Fig. 14. Effect of resource balancing on throughput (FPS) of MN inference on Alveo boards. Numbers at top of each bar indicate the maximum number of MN accelerator instances feasible to implement on a single board of the specified type. (*FPS estimated from per-SLR congestion of partitioned design.)

maximum number of accelerator instances our partitioner believes are feasible to implement, while for U280 and U250, it indicates actual implemented number of instances.

For U50, which is a relatively small card (two SLRs), both strategies yield the same estimated throughput and one accelerator instance utilizes most of the available resources. For U250, the largest card (four SLRs), the throughput of the two strategies is once again close, and each of them is able to produce a feasible implementation of two MN instances, which is expected given that the U250 is approximately twice the size of U50. The advantage of compression becomes apparent on the two intermediate-size boards, U200 and U280. On both, with the default FINN resource allocation, we are able to fit only one accelerator in the three available SLRs. In this experiment, our resource balancing flow is able to effectively choose between LUT/DSP and BRAM/URAM pairs for each layer, making the implementation of two MN instances on these cards feasible. However, the dual-MN designs are more congested than the default single-MN design and therefore their operating frequency is lower. We measure 78% FPS improvement on U280, and given the congestion of the design, a similar result is likely for U200. If we consider the number of SLRs utilized as a measure of cost, then resource balancing enables an increase in throughput per unit cost by moving from the larger U250 card to the smaller U200 or U280 with no loss in throughput.

6.4 Effect of Partitioning on Achievable Operating Frequency

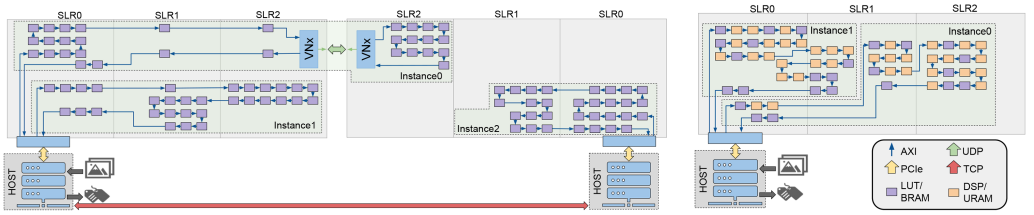
While many variables affect the achievable $F_{\max}$ for a given DFA, including target frequency, DFA structure, and congestion, we nevertheless attempt to quantify the contribution of our floorplanning flow to the achievable frequency. We compare builds of partitioned and non-partitioned but otherwise identical ResNet-50 and MobileNetV1 DFAs, on the same target FPGA. We execute each non-partitioned build twice—once with resets in SLR0 (the Vitis default), and once with resets controllers placed in a more central location (SLR1)—for each of five target frequencies. We report Vitis-reported post-routing frequencies.

Table 5 presents the results of our evaluation, which indicate significant speed-up can be achieved with partitioning over the Vitis default build strategy with monolithic kernels, and moderate speed-up from the hand-optimized Vitis reset strategy. There are no large differences in behavior between the two types of accelerator with regards to speed-up, but there is a relatively large difference between performance on U250 and U280, with the former benefitting more from partitioning relative to the reset-optimized Vitis strategy. We speculate this is due to the structure of the FPGA devices—the reset controller in SLR1 is more central on the 3-SLR U280 than on the 4-SLR U250. Congestion is not affected significantly: maximum per-SLR resource utilization is reduced on average by 4% for LUTs and 6% for BRAM.

These results indicate that the principal contributor to the speedup achieved through automatic partitioning on a single device is the optimization of reset infrastructure through the separation of

Table 5. F_{max} of Baseline and Partitioned DNN Accelerators

Accelerator	FPGA	#CUs	F_{max} Monolithic RSTC in SLR0 / SLR1	F_{max} Partitioned	Speedup
RN-50	U250	1	109/154	190	1.74/1.23
RN-50 (Fold 2×)	U280	1	123/208	217	1.76/1.04
MN	U250	2	110/142	210	1.90/1.48
MN	U280	1	152/215	241	1.58/1.12



(a) 3xMN in 2xU280 (DP+TMP, no resource balancing) (b) 2xMN in 1xU280 (balanced)

Fig. 15. Example partitioning results.

the DF pipeline into multiple kernels, which Vitis is better able to service. It is also worth noting that we observed an increased routing success rate when the design was partitioned. For example, in the case of the RN-50 implementation, three of five builds (with different target frequencies) failed when the default configuration was used, two of five failed in the reset-optimized case, but all builds succeeded when the DFA was partitioned. We also observe a reduction in total build runtime by approximately 10% when using partitioning.

Given the combination of multi-SLR target FPGAs, the high-congestion nature of these designs (high LUT and BRAM utilization) and the HLS design entry (FINN-generated C++ compiled with Vivado HLS), we consider the achieved frequencies of the monolithic (non-partitioned) designs adequate. With partitioning, these designs approach the performance of hand-crafted MPE accelerators. For reference, Xilinx Vitis AI MPE-based inference designs operate at 300 MHz on the same Alveo boards [63].

6.5 Discussion

Our evaluation indicates clear advantages from partitioning over the default Vitis build flow for FINN-generated DFAs, but does not paint a picture of the difficulty of partitioning these systems. Figure 15 illustrates the two most interesting designs in our evaluation—hybrid DP+TMP parallelism for three MobileNetV1 DFA instances in two Alveo U280s, and the compressed U280 design with two DFA instances. It is immediately obvious that manual partitioning of these systems is not feasible. Especially in high congestion areas, best illustrated by Instance 1 in Figure 15(b), the design must combine resource balancing and non-monotone partitions to meet congestion targets. No tiling placement patterns emerge in these designs, indicating that a global view of the entire multi-DFA system is better that a tiled design of individually partitioned DFAs.

Although a number of single- and multi-node DNN inference implementations exist, direct performance comparisons between these works is challenging due to differences in chosen topology, model quantization and sparsity, hardware optimization objectives, RTL versus HLS design, evaluation methodology and which performance metrics are reported. Table 6 compares our work

Table 6. Comparison to Single- and Multi-node FPGA Inference Accelerators

	Work	Paradigm	FPGA	Topology	Design	FPS	Latency (ms)	Quantization
Single-node	Ours	DF	Alveo U280	MobileNetV1	HLS	3741	2.3	w4a4
	Ours	DF	Alveo U250	ResNet-50	HLS	3127	1.7	w1a2
	[37]	DF	VU9P	ShuffleNet-v2	HLS	3321	n/a	w1a8
	[20]	DF	Stratix 10	MobileNetV1	RTL	5157	0.65	w16a16
	[64]	MPE	KU115	HCNet	RTL	1465	2.7	w16a16
	[30]	MPE	Arria 10	MobileNetV2	RTL	1050	6.2	w16a16
	[15]	MPE	Arria 10	ResNet-50	RTL	559	1.8	msfp-1s5e5m
Multi-node	Ours	DF	2 × Alveo U280	MobileNetV1	HLS	5923	2.2	w4a4
	Ours	DF	2 × Alveo U280	ResNet-50	HLS	3617	1.55	w1a2
	[3]	DF	3 × Stratix V	AlexNet	HLS	1,666.67	0.6	w1a2
	[66]	DF	4 × ADM-PCIE-8V3	ResNet-152	N/A	n/a	16.1	w16a16
	[54]	DF	3 × ZU19EG	Autoencoder	HLS	n/a	0.08	n/a
	[65]	MPE	4 × Virtex 7 690T	AlexNet	N/A	589.71*	104	w16a16
	[65]	MPE	4 × Virtex 7 690T	AlexNet	N/A	92.00	30.6	w16a16
	[26]	MPE	16 × ZCU102	AlexNet	HLS	3,225.81*	0.31	w16a16
	[48]	MPE	5 × AWS-F1	SqueezeNet	HLS	2,222.22	0.45	w16a16
	[48]	MPE	5 × AWS-F1	ResNet-18	HLS	476.00	2.1	w16a16

*Estimated from provided performance metrics.

with results from prior work on FPGA inference accelerators for ImageNet classification, with single-node results on the top part and multi-node ones at the bottom. Note that the table contains estimations for certain not-reported metrics, whereas other metrics cannot be estimated (e.g., latency for DF architectures cannot be derived from FPS) and are marked as not available (n/a). Among the single-node results, Hall et al. [20] report the highest throughput (close to 5.2 kFPS with 0.65 ms latency for MobileNetV1) by combining a DF-style architecture built of carefully optimized RTL components, yielding a high $F_{\text{clk}} = 430$ MHz. This is followed by our single-node results using HLS, achieving 3.7 kFPS with 2.3 ms for MobileNetV1, which outperforms the ShuffleNet-v2 implementation with JPEG compression by Nakahara et al. [37], as well as the MPE implementations with optimized RTL. We expect that combining our partitioning techniques with optimized RTL components would yield a further increase in performance for both single- and multi-node implementations, although this is left for future work. Among the multi-node FPGA implementations we surveyed, our two-node results provide the best-in-class measured performance, both in terms of highest FPS (5.9k FPS with MobileNetV1) and minimum latency (1.7 ms for ResNet-50).

Due to large variations in FPGA device sizes, number of nodes, and degree of quantization in Table 6, we provide additional comparisons on performance scalability of multi-FPGA inference in Table 7. With regard to throughput, which is the main focus of our optimization, only [26] is able to achieve super-linear scaling (Speed-up greater than 2) like our own results. When considering only the application of Elastic-DF model parallelism, our speed-up of 2.82 is lower than 3.43 for [26] (AlexNet). We note however that the mechanism by which speed-up is obtained in Reference [26] is through the circumvention of a DDR throughput bottleneck, whereas our own baselines have no such bottleneck, and we achieve speed-up through increased utilization of fabric resources, for which model parallelism is only a coarse-grained solution. Indeed, when considering compression as well, our speed-up from the baseline is 3.73, higher than that of Reference [26]. Unfortunately, we have to exclude Brainwave from our multi-node comparison, as detailed performance metrics are reported only for single-node instances.

Our analysis of the effect of partitioning on timing closure validates our initial assumption that fine-grained explicit dataflow is preferable to embedding the dataflow into a single Vitis kernel, and suggests that the typical approach of dataflow kernel design for Vitis, whereby dataflow is expressed internally to the kernel through the HLS DATAFLOW pragma, is counterproductive. We

Table 7. Comparison with State of the Art in Respect to Throughput and Latency Scaling for One or Two Nodes

Work	Paradigm	Topology	Two-node improvement	
			Throughput	Latency
Ours	DF	ResNet-50	2.06	1.78
	DF	MobileNetV1	2.82 (3.73)	1.00
[65]	MPE	AlexNet	1.5	1.00
	MPE	VGG	2.00	1.00
[26]	MPE	AlexNet	3.43*	3.43
	MPE	SqueezeNet	2.01*	2.01
[48]	MPE	AlexNet	1.75	1.75
[55]	DF	Autoencoder	n/a	2.16**

*Speed-up extrapolated from latency for MPE architectures.
**Extrapolated two-node from three-node performance.

believe based on our results that Vitis would be able to achieve higher quality of results by breaking up HLS dataflow regions into an explicit pipeline of kernels, which are assigned to SLRs by a partitioner such as ours. Alternatively, Vitis should place the reset controller as close as possible to the center of the FPGA to minimize the timing impact of reset signals.

6.5.1 Limitations. Our analysis has several limitations. Because we use an exact ILP solver for partitioning and resource selection, our approach may not be fast when applied to either very deep networks or very fine-grained designs where, e.g., every layer is exposed at the dataflow layer as many individual processing elements. The second limitation stems from our approach of favoring working prototypes and standard benchmarks over simulators or modelling—we are limited in our choice of DNNs, specifically to MLPerf DNNs that can be deployed to FPGA using a dataflow DNN compiler. At the time of writing, the quantized MobileNetV1 and ResNet-50 compiled with FINN were the only DNNs meeting our criteria. Additionally, our use of FINN carries over certain limitations in folding, which prevent us from scaling up these designs to more than two FPGAs. The third limitation relates to the optimality and repeatability of F_{max} on the platforms under analysis. While our configuration of Vitis is aimed at achieving maximum-effort timing closure, there is no guarantee that no higher-frequency implementations exist of any of the systems presented in this work. Due to the long duration of the implementation flow, we could not perform a comprehensive sweep of random seeds and other parameters affecting quality-of-results.

Finally, we do not provide mitigation for the potential impact of using UDP (unreliable) instead of TCP (reliable) as transport layer. We can however estimate the impact of UDP loss on the accuracy of the deployed models. If we consider one UDP error every 10 s (see our analysis in Section 4.3) and assume that a bit error produces a misclassification, then the drop in accuracy for, e.g., ResNet-50 TMP (3374 FPS), is $1/33,740 \approx 3 \times 10^{-5}$ on average. Even though multiple hops would increase the accuracy penalty linearly, the overall effect is negligible at any reasonable number of hops in a datacenter.

7 RELATED WORK

To address the computational demands of DNN inference, numerous customized FPGA accelerator architectures (Section 2) with both single- and multi-node implementations have been proposed. In this context, a number of algorithms and tools also explore the problems of partitioning, floorplanning and mapping of the workload to accelerators, similar to our Elastic-DF partitioner. In the following subsections, we present a selection of related work and carry out a qualitative

Table 8. Qualitative Comparison of Multi-node FPGA Inference Acceleration Schemes

Work	FPGA	Interconnect	Protocol	Communication*	Paradigm	Objective*	Partioning**
Ours	U250/U280	Ethernet @ 100 Gbps	UDP	P2P or host	DF	TR	inter-layer, ILP
[3]	Stratix V	MaxLink	N/A	P2P	DF	N/A	inter-layer, U
[66]	VU095	Ethernet @ 10 Gbps	TCP/IP	host	DF	T	inter-layer, DP
[55]	ZU19EG	Ethernet @ 100 Gbps	UDP	P2P	DF	T, L	inter-layer, G
[65]	XC7VX690T	Aurora @ 750 MB/s	N/A	P2P	MPE	T, L	inter-layer, DP
[26]	ZU9EG	Aurora @ 2.23 GB/s	N/A	P2P	MPE	L	intra-layer, ILP
[48]	AWS F1	PCIe	PCIe	host	MPE	T	inter-layer, GP
[9]	Stratix 10	Ethernet @ 40Gbps	custom	P2P	MPE	L	intra-layer, U

*P2P = peer-to-peer, T = throughput, L = latency, TR = throughput per resource.

**U = unspecified, ILP = integer linear programming, G = greedy, DP = dynamic programming, GP = geometric programming.

comparison to our own work for both single-node (multi-SLR) and multi-node (multi-FPGA) inference scenarios. Finally, since multi-node implementations, which are an important aspect of our efforts, are intricately dependent on the networking infrastructure, we include an overview of related work on FPGA cloud infrastructure and network stacks.

7.1 Partitioning Designs for Single-node, Multi-SLR FPGAs

Previous work on design partitioning for multi-SLR FPGAs is mostly focused on timing closure optimization. References [19, 38] make changes to the VPR place and route tool to minimize the number of wires crossing between SLRs in an effort to minimize timing impact. References [34, 35] further separate the placement flow into a coarse partitioning stage, which minimizes SLR crossings and a detail placement stage for each partition independently. All of these research efforts are general-purpose CAD tools and do not modify the underlying design. In Reference [61], a greedy partitioner is utilized to optimize the mapping of design buffers to physical OCM on multi-SLR devices. The partitioner utilizes size thresholds to decide whether to implement memories in distributed, Block or Ultra RAM on Xilinx VU9P devices. Our Elastic-DF ILP partitioner takes these concepts further and adds a multi-FPGA dimension—we provide simultaneously SLR crossing minimization, heterogeneous inter-SLR (including inter-FPGA) communication bandwidth limitation, and congestion management through Block/Ultra RAM and LUT/DSP utilization trade-off. Also, to our knowledge, no other partitioning tool provides the anchoring capability that enables us to implement transparent model-parallelism. While we focus on DNN dataflow accelerators in this work, our partitioner is sufficiently generic to be utilized with any FPGA design.

7.2 Multi-node FPGA DNN Inference

A number of prior works describe multi-node DNN inference implementations for FPGAs. Despite the shared multi-node aspect, these works differ substantially in terms of optimization objectives, acceleration paradigm, model quantization, accuracy on dataset, evaluation methodology, and reported metrics, which makes direct performance comparisons difficult. To supplement the quantitative comparison in Section 6.5 with a qualitative one, we now offer a summary of key aspects related to multi-node implementations in Table 8. We structure our discussion around the partitioning and network connectivity aspects, and how Elastic-DF differs in these regards.

7.2.1 Partitioning. The method for deciding the allocation of DNN computations to multiple FPGAs is dependent on the particular acceleration paradigm (Section 2.1), with various partitioning granularities, methods, and optimization objectives possible. The three final columns of Table 8 summarize these properties for the related work. Taking these properties into account, one can construct a performance model for predicting the desired performance metric based

on allocation decisions, then apply mathematical optimization methods to search for a good partitioning. MPE-based approaches typically instantiate identical, maximum-sized hardware on each available FPGA, then partition the DNN inference graph to these accelerators with latency and/or throughput as optimization objectives. For MPE designs where weights and activations are stored in off-chip memory, the available memory bandwidth often becomes the bottleneck and is the key consideration for partitioning. Super-LIP [26] distributes (parts of) layers to MPEs on different nodes to achieve super-linear latency reduction by moving traffic from the memory bus to inter-FPGA links leveraging an ILP solver. Zhang et al. [65] describe a similar approach, but provide options to run their partitioner to target either latency or throughput as the objective. For this they use a dynamic programming approach. Brainwave [9] mentions partitioning the DNN inference graph over a number of mega-SIMD MPEs (called NPUs) replicated over all nodes and fixed in size, but provide no further details on how the partitioning works or how the latency reduction scales with the number of nodes. Shan et al. [48] partition a DNN workload to FPGAs communicating over PCIe links in an AWS-F1 instance. Their partitioner constrains the resource utilization of each FPGA and allows non-monotonic partitions.

For DF-based multi-node acceleration, there is greater focus on inter-layer partitioning to minimize bisection bandwidth and staying within each FPGA's resource constraints while specializing each layer's compute engine. Baskin et al. [3] and Tarafdar et al.[55], the latter using a greedy algorithm, consider splitting DF architectures across multiple FPGAs to be able to accommodate larger models such as ResNets, but do not describe any toolflow support for further un-folding for scaling up the performance. Zhang et al. [66] describe a dynamic programming approach to partitioning that maximizes throughput, setting the unrolling factors for each layer while respecting the resource budget, but provide no details on the implementation, unrolling granularities or throughput balancing aspects. We also note FPDeep by Geng et al. [62], which proposes a multi-FPGA DNN training framework. Although DNN training involves quite different data dependencies compared to inference, FPDeep provides automatic partitioning of the training task graph to MPEs through a multi-step approximation algorithm whereby the optimization goal is throughput, leveraging both model- and data-parallelism, but which does not take inter-FPGA communication throughput requirements into account. FPDeep also provides resource balancing as a fine-tuning step, after the partitioning is completed, but only allows trade-off between two classes of resources.

Elastic-DF differs from these works in two key aspects. The first is a fundamental difference in approach: in contrast to existing solutions that optimize FPS or latency through partitioning and resource allocation, Elastic-DF starts from an existing folding solution that dictates latency, allowing multiple DFA instances segmented across multiple FPGAs and optimizing for the performance per unit area metric. To maximize this metric, we combine partitioning and resource balancing into a single step, support non-monotone partitions, and allow the user full control of the resource balancing options for each DNN node.

The second distinction is that none of these prior works consider the combination of multi-SLR and multi-FPGA partitioning, which our analysis indicates to be key to achieving good performance results for large DC-class FPGAs. We also note that Elastic-DF can in principle perform intra-layer partitioning if the underlying DFA compiler would expose a finer grained layer structure, e.g., individual PEs, although this may increase the partitioner runtime significantly, since we utilize an exact ILP solver.

7.2.2 Network Connectivity. We consider three aspects of network connectivity in Table 8: the physical interconnect, protocol and communication style used for distributed inference, all of which impact performance, scalability and portability of the solution. We differentiate between DF and MPE implementations, and regarding the partitioning, what the optimization objectives

are, at what boundaries the partitioner operates, and the type of algorithm used. Shan et al. [48] only consider communication over PCIe for an AWS-F1 instance with multiple FPGAs. Several works [3, 26, 65] use direct P2P serial links for communication, which is low overhead and requires no protocol, but limits scalability in datacenter scenarios. Others, including Brainwave [9], AIgean [55], Zhang et al. [66] and our own work make use of Ethernet, which offers greater scalability and compatibility. Among these works, [66] uses host-initiated communications, in contrast to the others where accelerators initiate direct P2P communications to avoid being bottlenecked by the host CPU. Brainwave [9] leverages a custom protocol, whereas our work uses standard UDP/IP. While customized protocols lead may lead to more efficient implementations, standardized protocols enables easier interoperability.

7.3 FPGA Cloud Infrastructure and Network Stacks

Several data center operators provide FPGAs as a service, including AWS [2], Huawei [23], Nimbix [39], and Microsoft Azure [8]. AWS and Nimbix do not currently provide node-to-node connectivity between FPGAs. Azure offers 40 Gbps Ethernet connectivity per node, where the communication is handled over a customized UDP protocol. Huawei has dedicated 300 Gbps optical links between nodes with a mesh interconnect; however, we did not have access to detail on how network abstractions are provided to the user and what form of communication primitives have been made available. In the FPGA research community, we are aware of two efforts for distributed FPGA infrastructure. IBM's CloudFPGA [1] is a highly customized platform that integrates 32 FPGA cards per chassis, with full cross connectivity provided amongst the nodes through a crossbar switch. FPGAs communicate directly with each other using MPI primitives over 10 Gbps Ethernet connections. Galapagos [56] supports 10, 40, and 100 Gbps Ethernet with either generic UDP or TCP/IP transport protocols, plus MPI abstraction for communication. In principle, any of these cloud infrastructure options would be suitable for running our distributed CNN inference designs, but XACC is currently the only publicly accessible infrastructure that we are aware of that offers explicit support for peer-to-peer FPGA communication.

With regard to the implementation of peer-to-peer links, several papers have proposed FPGA network stacks [4, 13, 14, 24]. Work at ETH Zurich [49, 50] led to a 10 Gbps TCP stack written in C/C++ harnessing Vivado HLS and later to Limago [46], the first complete description of an open-source implementation of a TCP/IP stack operating at 100 Gb/s. Intel **Inter-Kernel Links (IKL)** [52] provides light-weight direct Ethernet connectivity between FPGAs at 40 Gbps. Our VNx focuses on efficiency and is able to achieve equivalent resource utilization with IKL at higher speed by implementing a UDP stack based on Limago.

8 CONCLUSION AND FUTURE WORK

We proposed Elastic-DF as a methodology and set of tools that efficiently map DFAs to FPGAs. We demonstrate the applied partitioner enables super-linear scaling of throughput for both MobileNetV1 and ResNet-50-v1.5, by leveraging model parallelism and direct FPGA communication over 100 Gbps Ethernet networked nodes. Further throughput increases are possible when performing tightly-coupled partitioning and resource allocation. We also observe that automatic partitioning is effective for working around platform-specific timing closure challenges for large multi-SLR accelerators. In most cases, the automatic partitioning flow enables the identification of the maximum number of accelerator instances and allocation of layers to SLRs and devices on a given platform in minutes, which is negligible compared to the typical duration of an FPGA implementation flow. Compared to other similar approaches, such as Microsoft's Brainwave, we differentiate most importantly through more flexible scaling, resulting from the fine-grained dataflow generated by Elastic-DF and FINN instead of using a fixed NPU architecture.

This work is just the start of a larger effort. We intend to explore the scalability to much deeper CNNs, investigate effects of different network protocols, e.g., TCP and connectivity topologies on the scalability and DFA density. Given the generality of the approach, future work could focus on the partitioning of more generic compute graphs, outside the scope of dataflow inference. Finally, a tighter integration with high-level synthesis tools could allow partitioning to be performed during accelerator compilation, providing additional mechanisms for performance improvement.

APPENDIX

A ILP FORMULATION OF THE ELASTIC-DF PARTITIONER

A.1 Input Definition

We first mathematically define the dataflow compute graph and the target FPGA platform.

Dataflow Graph. The task graph $T = (T^n, T^e)$ carries the information regarding the computations to be implemented, where T^n represents the set of DF nodes, e.g., DNN layers, and T^e represents the set of connections between nodes.

Each node $p \in T^n$ is defined for the purposes of partitioning by its potential implementations and their respective resource utilization profiles. We denote the set of node resource types as U^n and $d = |U^n|$ is the total number of node resource types. For FPGA implementations, typical resource types are LUTs, FFs, DSPs, BRAMs and URAMs so $d = 5$. Each possible node implementation $v \in p$ is therefore a vector in $\mathbb{Q}^d$ or $\mathbb{N}_0^d$ specifying the utilization of each resource type for that implementation. For example, we can have a version using DSPs for a certain computation and an alternative version using LUTs. We denote t^n_{p,v,u^n} the requirement of resource type $u^n \in U^n$ of version $v \in p$ of task $p \in T^n$.

Each task edge is an ordered pair $k = \{p, q\} \in T^e$, which establishes a connection from task node p to task node q, i.e., the output of task node p is the input of task node q. We denote the set of connection resource types as U^e. For multi-FPGA accelerators, connections between task nodes can be established through dedicated wires if task nodes reside on the same FPGA or through shared chip-to-chip connectivity if nodes reside on different FPGAs. The number of wires and off-chip throughput define the set of resource types in this example. We denote t^e_{k,u^e} the edge requirement of resource type $u^e \in U^e$ associated with task edge $k \in T^e$.

Compute Platform. The graph $C = (C^n, C^e)$ describes the target platform. Each node in C^n represents a compute unit we can assign tasks to, for example, for Multi SLR FPGA devices it can be a SLR, or otherwise a pBlock or even a whole FPGA device. Each compute node $i \in C_n$ provides a certain amount c^n_{i,u^n} of each resource type $u^n \in U^n$. Each edge in C^e establishes a connection between compute nodes i and j and provides up to c^e_{i,j,u^e} of each connection resources type $u^e \in U^e$. For example, a connection between SLRs residing on the same FPGA may provide a large number of wires (SLL) but no off-chip throughput, whereas a connection between SLRs on different FPGAs will provide no wires and some off-chip throughput.

A.2 Constraining the Partition

To be able to set ratios of utilization for the compute resources a limit $l_{u^n} \in [0, 1]$ is provided for each resource type $u^n \in U^n$. Average utilization constraints of the form: $Avg(u^n_0, u^n_1, u^n_2, ..) < l$ are defined by a list A^u of tuples $\{l, M\}$, where $l \in [0, 1]$ is the average utilization limit and $M \subseteq U^n$ is the subset of resource types included in the average.

Anchoring constraints are specified through two lists of tuples: A^a and A^r corresponding to absolute and relative anchors, respectively. Each tuple $\{p, N\} \in A^a$ restrict a task $p \in T^n$ to be placed in a subset of compute nodes $N \subseteq C^n$. Each tuple $\{p, q\} \in A^r$ forces task $p \in T^n$ to be in the same compute node as task $q \in T^n$.

A.3 Optimizing the Partition

The partitioning algorithm described herein maps the task graph to the compute graph, within the constraints specified. In addition, we want to optimize placement and/or connections of the task nodes. For this, we use an additional matrix, the connection cost matrix C^c. The element $c^c_{i,j} \in C^c$ is the cost associated to the connection between compute node i and j.

The partitioning is expressed as an Integer Linear Program (ILP). For the formulation of the problem, we use two sets of binary variables, the edge map M^e and the node map M^n. The edge map variables $m^e_{i,j,k} \in M^e$ state whether the task dependency k connects a task node mapped to compute node i with a task node mapped to compute node j. The other set of auxiliary variables are the node map variables $m^n_{i,p,v} \in M^n$, that state whether the version v of task node p is assigned to the compute node i.

Finally, the ILP formulation is

$$\text{minimize} \sum_{i \in C^n} \sum_{j \in C^n} \sum_{k \in T^e} m^e_{i,j,k} * c^c_{i,j}, \tag{1}$$

subject to:

$$\forall p \in T^n : \sum_{i \in C^n} \sum_{v \in p} m^n_{i,p,v} = 1, \tag{2}$$

$$\forall k = \{p,q\} \in T^e, \forall i \in C^n : \sum_{j \in C^n} m^e_{i,j,k} = \sum_{v \in p} m^n_{i,p,v}, \tag{3}$$

$$\forall k = \{p,q\} \in T^e, \forall j \in C^n : \sum_{i \in C^n} m^e_{i,j,k} = \sum_{v \in q} m^n_{j,q,v}, \tag{4}$$

$$\forall i \in C^n, \forall u^n \in U^n : \sum_{p \in T^n} \sum_{v \in p} m^n_{i,p,v} * t^n_{p,v,u^n} \leq c^n_{i,u^n} * l_{u^n}, \tag{5}$$

$$\forall \{i,j\} \in C^e, \forall u^e \in U^e : \sum_{k \in T^e} m^e_{i,j,k} * t^e_{k,u^e} \leq c^e_{i,j,u^e}. \tag{6}$$

Here, Equation (2) forces each task node to be placed once and only once. Constraint Equations (3) and (4) create the link between M^e and M^n, that is, these variables are not independent. For each task edge $k = \{p,q\} \in T^e$, there are compute nodes i and j such that $m^e_{i,j,k} = 1$. Then, by definition there must be $v_0 \in p$ and $v_1 \in q$ such that $m^n_{i,p,v_0} = 1$ and $m^n_{j,q,v_1} = 1$, respectively. Resource limitations are set by Equations (5) and (6) for node resources and connection resources, respectively.

Applying anchors. The formulation for these constraints is

$$\forall \{p,N\} \in A^a : \sum_{i \in N} \sum_{v \in p} m^n_{i,p,v} = 1, \tag{7}$$

$$\forall \{p,q\} \in A^r, \forall i \in C^n : \sum_{v \in p} m^n_{i,p,v} = \sum_{v \in q} m^n_{i,q,v}. \tag{8}$$

Applying average utilization constraints. Defining C^n_M as the subset of nodes in C^n for which $u^n > 0$ for $u^n \in M$, the formulation for these constraints is

$$\forall \{l,M\} \in A^u, \forall i \in C^n_M : \frac{1}{|M|} \sum_{u^n \in M} \frac{1}{c^n_{i,u^n}} \sum_{p \in T^n} \sum_{v \in p} m^n_{i,p,v} * t^n_{p,v,u^n} \leq l. \tag{9}$$

REFERENCES

[1] F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, and S. Paredes. 2017. An FPGA platform for hyperscalers. In *Proceedings of the IEEE 25th Annual Symposium on High-performance Interconnects (HOTI'17)*. 29–32. https://doi.org/10.1109/HOTI.2017.13

[2] Amazon AWS. 2018. Retrieved from https://aws.amazon.com/ec2/instance-types/f1/.

[3] Chaim Baskin, Natan Liss, Evgenii Zheltonozhskii, Alex M. Bronstein, and Avi Mendelson. 2018. Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*. IEEE, 162–169.

[4] Giuseppe Bianchi, Michael Welzl, Angelo Tulumello, Giacomo Belocchi, Marco Faltelli, and Salvatore Pontarelli. 2018. A fully portable TCP implementation using XFSMs. In *Proceedings of the ACM SIGCOMM Conference on Posters and Demos*. ACM, 99–101.

[5] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfig. Technol. Syst.* 11, 3 (2018), 1–23.

[6] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. 2020. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq Ultrascale+. In *Proceedings of the International Conference on Field-Programmable Technology*.

[7] Salvatore Calǵ, Grzegorz Korcyl, and Piotr Korcyl. 2020. Using Xilinx Alveo accelerators for lattice QCD. In *Proceedings of the Asia-Pacific Intertional Symposium on Lattice Field Theory*, Vol. 2020.

[8] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim et al. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–13.

[9] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman et al. 2018. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[10] ONNX Community. [n.d.]. ONNX: Open Neural Network Exchange. Retrieved from https://github.com/onnx.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. IEEE, 248–255.

[13] Li Ding, Ping Kang, Wenbo Yin, and Linli Wang. 2016. Hardware TCP offload engine based on 10-Gbps ethernet for low-latency network communication. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'16)*. IEEE, 269–272.

[14] Hamish Fallside, M. John, and S. Smith. 2000. Internet connected FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 289–290.

[15] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 1–14.

[16] Jeremy Fowers, Kalin Ovtcharov, Michael K. Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi et al. 2019. Inside project Brainwave's cloud-scale, real-time AI processor. *IEEE Micro* 39, 3 (2019), 20–28.

[17] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. 2018. ReBNet: Residual binarized neural network. In *Proceedings of the IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. IEEE, 57–64.

[18] ETH Zurich Systems Group. [n.d.]. Vitis with 100 Gbps TCP/IP Network Stack. Retrieved from https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP.

[19] Andre Hahn Pereira and Vaughn Betz. 2014. Cad and routing architecture for interposer-based multi-fpga systems. In *Proceedings of the ACM/SIGDA international symposium on Field-programmable gate arrays*. 75–84.

[20] Mathew Hall and Vaughn Betz. 2020. HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs. Retrieved from https://arXiv:2007.10451.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. Retrieved from https://arXiv:1512.03385.

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. Retrieved from http://arxiv.org/abs/1704.04861.

[23] Huawei. 2019. Retrieved from https://www.huaweicloud.com/en-us/product/fcs.html.

[24] Yong Ji and Qing-Sheng Hu. 2011. 40Gbps Multi-Connection TCP/IP Offload Engine. In *Proceedings of the International Conference on Wireless Communications and Signal Processing (WCSP'11)*. IEEE, 1–5.
[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. Retrieved from https://arXiv:1807.05358.
[26] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Trans. Embed. Comput. Syst.* 18, 5s (2019), 1–23.
[27] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*. ACM, 1–12.
[28] Christoforos Kachris. 2018. *Performance evaluation of InAccel ML scalable suite*. Technical Report. InAccel. Retrieved from https://www.inaccel.com/wp-content/uploads/inaccel_white_paper.pdf.
[29] Vinod Kathail. 2020. Xilinx vitis unified software platform. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 173–174.
[30] Justin Knapheide, Benno Stabernack, and Maximilian Kuhnke. 2020. A high throughput MobileNetV2 FPGA implementation based on a flexible architecture for depthwise separable convolution. In *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (FPL'20)*. IEEE, 277–283.
[31] Mairin Kroes, Lucian Petrica, Sorin Cotofana, and Michaela Blott. 2020. Evolutionary bin packing for memory-efficient dataflow inference acceleration on FPGA. In *Proceedings of the International Conference on Genetic and Evolutionary Computation (GECCO'20)*.
[32] Xilinx Research Labs. [n.d.]. FINN Dataflow Accelerator Examples. Retrieved from https://github.com/Xilinx/finn-examples.
[33] Xilinx Research Labs. [n.d.]. FINN: Dataflow compiler for QNN inference on FPGAs. Retrieved from https://github.com/Xilinx/finn.
[34] Fubing Mao, Wei Zhang, Bo Feng, Bingsheng He, and Yuchun Ma. 2016. Modular placement for interposer based multi-FPGA systems. In *Retrieved from International Great Lakes Symposium on VLSI (GLSVLSI'16)*. IEEE, 93–98.
[35] Kevin E. Murray and Vaughn Betz. 2015. HETRIS: Adaptive floorplanning for heterogeneous FPGAs. In *Proceedings of the International Conference on Field Programmable Technology (FPT'15)*. IEEE, 88–95.
[36] Naif Tarafdar, Giuseppe Di Guglielmo, Philip C. Harris, Jeffrey D. Krupa, Vladimir Loncar, Dylan S. Rankin, Nhan Tran, ZhenbinWu, Qianfeng Shen, and Paul Chow. [n.d.]. AIgean: An Open Framework for Machine Learning on a Heterogeneous Cluster. Retrieved from https://indico.cern.ch/event/924283/contributions/4105333/attachments/2154984/3634529/aigean_fastml.pdf.
[37] Hiroki Nakahara, Zhiqiang Que, and Wayne Luk. 2020. High-Throughput convolutional neural network on an FPGA by customized JPEG compression. In *Proceedings of the IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'20)*. IEEE, 1–9.
[38] Ehsan Nasiri, Javeed Shaikh, Andre Hahn Pereira, and Vaughn Betz. 2015. Multiple dice working as one: CAD flows and routing architectures for silicon interposer fpgas. *IEEE Trans. Very Large Scale Integr. Syst.* 24, 5 (2015), 1821–1834.
[39] Nimbix. 2019. Retrieved from https://www.nimbix.net/alveo.
[40] Alessandro Pappalardo. [n.d.]. Brevitas: Quantization-Aware Training in PyTorch. Retrieved from https://github.com/Xilinx/brevitas.
[41] Lucian Petrica. [n.d.]. Elastic-DF partitioner and FINN integration. Retrieved from https://github.com/Xilinx/finn-experimental/blob/main/src/finn/util/platforms.py.
[42] Lucian Petrica. [n.d.]. Quantized ResNet-50 Dataflow Acceleration on Alveo. Retrieved from https://github.com/Xilinx/ResNet50-PYNQ.
[43] Lucian Petrica, Tobias Alonso, Mairin Kroes, Nicholas Fraser, Sorin Cotofana, and Michaela Blott. 2020. Memory-Efficient Dataflow Inference for Deep CNNs on FPGA. Retrieved from https://arXiv:2011.07317.
[44] Xilinx University Program. [n.d.]. Xilinx Adaptive Compute Cluster (XACC) Program. Retrieved from https://www.xilinx.com/support/university/XUP-XACC.html.
[45] Mario Ruiz. [n.d.]. XUP Vitis Network Example (VNx). Retrieved from https://github.com/Xilinx/xup_vitis_network_example.
[46] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-based Open-source 100 GbE TCP/IP Stack. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. IEEE, 286–292. https://doi.org/10.1109/FPL.2019.00053
[47] Kirk Saban. 2012. Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency. https://www.xilinx.com/support/documentation/white_papers/wp380_Stacked_Silicon_Interconnect_Technology.pdf.

[48] Junnan Shan, Mihai T. Lazarescu, Jordi Cortadella, Luciano Lavagno, and Mario R. Casu. 2021. CNN-on-AWS: Efficient allocation of multi-kernel applications on Multi-FPGA platforms. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 40, 2 (2021), 301–314. DOI: 10.1109/TCAD.2020.2994256

[49] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10 Gbps TCP/IP stack architecture for reconfigurable hardware. In *Proceedings of the IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 36–43.

[50] David Sidler, Zsolt István, and Gustavo Alonso. 2016. Low-latency TCP/IP stack for data center applications. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. IEEE, 1–4.

[51] IEEE Computer Society. 2014. IEEE standard for ethernet-amendment 2: Physical layer specifications and management parameters for 100 Gb/s operation over backplanes and copper cables. IEEE Std 802.3 bj-2014 (Amendment to IEEE Std 802.3-2014). Retrieved from https://standards.ieee.org/standard/802-3bj-2014.html.

[52] Roberto Dicecco Susanne M. Balle, Mark Tetreault. 2020. Inter-Kernel Links for Direct Inter-FPGA Communication. Retrieved from https://www.intel.com/content/dam/www/programmable/us/en/others/literature/wp/wp-01305-inter-kernel-links-for-direct-inter-fpga-communication.pdf.

[53] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[54] Naif Tarafdar, Giuseppe Di Guglielmo, Philip C. Harris, Jeffrey D. Krupa, Vladimir Loncar, Dylan S. Rankin, Nhan Tran, Zhenbin Wu, Qianfeng Shen, and Paul Chow. 2020. AIgean: An open framework for machine learning on heterogeneous clusters. In *Proceedings of the IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'20)*. IEEE, 239–239.

[55] Naif Tarafdar, Giuseppe Di Guglielmo, Philip C. Harris, Jeffrey D. Krupa, Vladimir Loncar, Dylan S. Rankin, Nhan Tran, Zhenbin Wu, Qianfeng Shen, and Paul Chow. 2020. AIgean: An open framework for machine learning on heterogeneous clusters. In *Proceedings of the 6th International Workshop on Heterogeneous High-performance Reconfigurable Computing*. IEEE.

[56] Naif Tarafdar, Nariman Eskandari, Varun Sharma, Charles Lo, and Paul Chow. 2018. Galapagos: A full stack approach to FPGA integration in the cloud. *IEEE Micro* 38, 6 (2018), 18–24.

[57] Haroldo G. Santos and Tulio A. M. Toffolo. [n.d.]. The Python MIP Package. Retrieved from https://www.python-mip.com/.

[58] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–74.

[59] Yaman Umuroglu and Magnus Jahre. 2017. Streamlined deployment for quantized neural networks. Retrieved from http://arxiv.org/abs/1709.04060.

[60] Stylianos I. Venieris and Christos-Savvas Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. IEEE, 40–47.

[61] Nils Voss, Pablo Quintana, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. 2019. Memory mapping for multi-die FPGAs. In *Proceedings of the IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*. IEEE, 78–86.

[62] Tianqi Wang, Tong Geng, Ang Li, Xi Jin, and Martin Herbordt. 2020. FPDeep: Scalable acceleration of CNN training on deeply-pipelined FPGA clusters. *IEEE Trans. Comput.* 69, 8 (2020), 1143–1158. https://doi.org/10.1109/TC.2020.3000118

[63] Xilinx. [n.d.]. Vitis AI Model Zoo. Retrieved from https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo.

[64] Xiaoyu Yu, Yuwei Wang, Jie Miao, Ephrem Wu, Heng Zhang, Yu Meng, Bo Zhang, Biao Min, Dewei Chen, and Jianlin Gao. 2019. A data-center FPGA acceleration platform for convolutional neural networks. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. IEEE, 151–158.

[65] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 326–331.

[66] Wentai Zhang, Jiaxi Zhang, Minghua Shen, Guojie Luo, and Nong Xiao. 2019. An efficient mapping approach to large-scale DNNs on multi-FPGA architectures. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'19)*. IEEE, 1241–1244.

Received January 2021; revised April 2021; accepted June 2021