TAPA-CS: Enabling Scalable Accelerator Design on Distributed HBM-FPGAs

Neha Prakriya
nehaprakriya@cs.ucla.edu
UCLA
Los Angeles, USA

Yuze Chi
chiyuze@cs.ucla.edu
UCLA
Los Angeles, USA

Suhail Basalama
basalama@cs.ucla.edu
UCLA
Los Angeles, USA

Linghao Song
linghaosong@cs.ucla.edu
UCLA
Los Angeles, USA

Jason Cong
cong@cs.ucla.edu
UCLA
Los Angeles, USA

Abstract

Despite the increasing adoption of FPGAs in compute clouds, there remains a significant gap in programming tools and abstractions which can leverage network-connected, cloud-scale, multi-die FPGAs to generate accelerators with high frequency and throughput. We propose TAPA-CS, a task-parallel dataflow programming framework which automatically partitions and compiles a large design across a cluster of FPGAs while achieving high frequency and throughput. TAPA-CS has three main contributions. First, it is an open-source framework which allows users to leverage virtually "unlimited" accelerator fabric, high-bandwidth memory (HBM), and on-chip memory. Second, given as input a large design, TAPA-CS automatically partitions the design to map to multiple FPGAs, while ensuring congestion control, resource balancing, and overlapping of communication and computation. Third, TAPA-CS couples coarse-grained floorplanning with interconnect pipelining at the inter- and intra-FPGA levels to ensure high frequency. FPGAs in our multi-FPGA testbed communicate through a high-speed 100Gbps Ethernet infrastructure. We have evaluated the performance of TAPA-CS on designs, including systolic-array based CNNs, graph processing workloads such as page rank, stencil applications, and KNN. On average, the 2-, 3-, and 4-FPGA designs are 2.1×, 3.2×, and 4.4× faster than the single FPGA baselines generated through Vitis HLS. TAPA-CS also achieves a frequency improvement between 11%-116% compared with Vitis HLS.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651347

ACM Reference Format:
Neha Prakriya, Yuze Chi, Suhail Basalama, Linghao Song, and Jason Cong. 2024. TAPA-CS: Enabling Scalable Accelerator Design on Distributed HBM-FPGAs. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651347

1 Introduction

In the big data era, there has been an exponential rise in the demand for scalable, cheap, and high performance acceleration. FPGAs have emerged as a promising solution to counter the breakdown of Dennard's scaling [22, 33] due to their reconfigurability and low power consumption. One of the greatest successful demonstrations is Microsoft's Catapult project which sped up the Bing Search engine using Stratix V FPGAs, achieving a 95% increase in throughput with a minimal power consumption increase of only 10% [51]. Microsoft also displayed the use of FPGAs for accelerating DNN inference and data compression in their servers [25, 30, 35, 36, 48]. Today, other major players such as Amazon [4, 5], Alibaba [1], Baidu [7], and Huawei [11] also use FPGAs, and offer them as a service in their cloud.

High-level synthesis (HLS) tools like Vitis HLS [15] and Intel HLS [12] raise the abstraction level for programming individual FPGAs in the cloud from RTL to C++/OpenCL. This allows the programmer to have little to no knowledge of the underlying hardware and cycle accuracy. While these tools deliver great results, they are limited to programming a single FPGA. At the same time, with the gradual end of Moore's law, accelerator designs are becoming larger than ever before, and require more programmable logic and memory than that available on a single FPGA device [47]. Our goal is to support the network-connected devices shown in Figure 1. Utilizing such multi-FPGA setups requires careful consideration for efficient workload distribution to ammortize the cost of inter-FPGA communication.

The design of modern FPGAs adds a new layer of complexity. FPGA architectures have varying interconnection

Method	HLS	Ethernet	Floorplanning	Interconnect Pipelining	Topology-Aware	Automatic Partitioning	Hardware Execution	Generalizable	Fmax (MHz)
FPGA'12[34]	×	×	×	×	×	×	×	✓	85
Simulation-based [42, 44, 55]	×	×	×	×	×	✓	×	✓	-
Virtualization-based [28, 61–63]	✓	✓	×	×	×	✓	✓	✓	100-300
CNN/DNN [17, 19, 20, 41, 57, 64, 65]	✓	✓	×	×	×	✓	✓	×	240
TAPA-CS (Ours)	✓	✓	✓	✓	✓	✓	✓	✓	300

Table 1. Comparison of TAPA-CS and existing methods providing scale-out acceleration across multiple FPGAs.

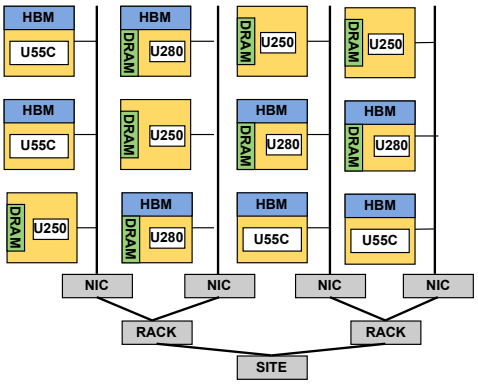

Figure 1. Network-Connected FPGAs

support (PCIe and Ethernet-based ports), different types of memory bandwidth and capacity (on-chip BRAM, off-chip DRAM/HBM), programmable logic units organized into multiple dies (i.e., chiplets), and degrees of data transfer cost (on-die, cross-die, cross-chip). Consider the example of the Xilinx/AMD Alveo U55C cards, organized as three chiplets. This card supports 2 ports offering 100Gbps bandwidth for networking. It also features an HBM with 16GB capacity exposing a bandwidth of 460GBps. The on-chip memory provides a high bandwidth of 35TBps but has a small capacity of 43MB [2]. We introduce details of modern FPGA architecture in Section 2.

An expert designer will consider all these factors when designing and partitioning their kernel code across chips. However, manual workload partitioning is inefficient as the design gets larger. Therefore, despite the advances in CAD tools which allow the user to program a single *FPGA in the cloud*, there is a significant lack of programming tools which can target *multiple FPGAs* potentially at the cloud scale. Such a framework should take as input large workloads from the user and automatically partition it across multiple devices efficiently. We identify the following three main challenges to address when developing such frameworks for cloud-scale FPGAs:

1. Need a lightweight inter-FPGA communication infrastructure which enables reliable and high speed data transfers.
2. Need to partition and map application code efficiently keeping in mind factors like compute-load balancing, network topology, the varying cost of on- and off-chip communication.
3. Need to ensure high design frequency.

Several prior works have attempted to leverage the networking capabilities available in modern FPGAs [3, 32, 39, 40, 52, 53, 56]. These prior works differ in the achievable data transfer throughput (10-90GBps), orchestration of data transfers (host/FPGA), and the resource overheads. We compare these methods in detail in Section 6.

We compare initial efforts in addressing Challenges 2 and 3 in Table 1. Simulation-based tools [42, 44, 55] enable rapid prototyping but are not substitutes for real hardware execution. Prior work such as [17, 19, 20, 41, 57, 64, 65] proposed CNN/DNN partitioning across FPGAs, but are not generalizable to different workloads. Other works such as [34] leverage latency-insensitivity (discussed in Section 4.3) to partition the design across FPGAs but expect the user to provide module-to-FPGA mappings in RTL, and perform simulation-based experiments. Recent virtualization-based work [61–63] also leverages latency-insensitive design to partition the workload, but virtualizes the FPGA by creating pre-placed and pre-routed static regions to which user logic is mapped. Most prior works take advantage of the networking capabilities available in modern FPGAs, but do not formulate their design partitioners in a way that minimizes and hides this latency. We find that none of these prior works consider coupling intelligent floorplanning of the compute modules and interconnect pipelining with HLS compilation. This step is crucial in achieving designs with high frequency as we discuss in Section 2. Also, none of the prior works consider the topology of the networked-FPGA infrastructure. This might result in the suboptimal mapping of compute modules to devices, and is a major hurdle in the scalability of the tool beyond two FPGAs. We discuss more details of prior works in Section 6.

To this end, we propose TAPA-CS which takes as input any large-scale dataflow workload expressed in C/C++, and automatically partitions and maps it to a cluster of modern FPGAs during HLS compilation. TAPA-CS couples the process of intra- and inter-FPGA floorplanning with interconnect pipelining to ensure high frequency. TAPA-CS is built upon the latest progress in dataflow-based FPGA HLS design tools [26, 37, 38]. Our main contributions are as follows:

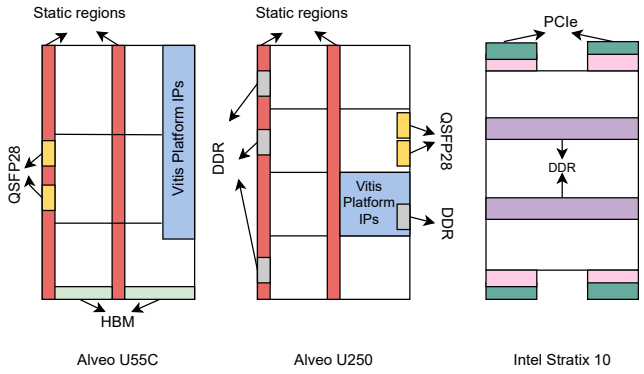

Figure 2. Architecture examples of modern FPGAs.

1. Integrate two layers of floorplanning (inter- and intra-FPGA) and interconnect pipelining with HLS compilation using an Integer Linear Programming (ILP)-based resource allocator which takes into account network topology and the internal FPGA chip layout.
2. Utilize the latency-insensitive nature of the dataflow design to partition it across devices, allowing us flexibility in implementing the inter-FPGA communication.
3. Raise the abstraction level of programming cloud-scale FPGAs by hiding the cluster complexity.
4. Test TAPA-CS on different applications ranging from systolic-array based CNNs, stencil designs, Page Rank, and KNN on 2-8 FPGAs. We achieve an average throughput increase of 2.1×, 3.2×, and 4.4× using 2, 3, and 4 FPGAs. We also improve the design frequency between 11-116% compared with Vitis HLS.
5. Discuss the computation to communication trade-offs of each benchmark and how they impact the scalability to multiple nodes containing up to 8 FPGAs.

2 Background

FPGAs consist of programmable logic organized in the form of a 2-D grid. This programmable region consists of look-up tables (LUTs) which can implement the truth table for any 6-input function. In recent years, FPGAs have also come to include several hard IPs such as PCIe IPs, DDR/HBM controllers, and other platform-specific IPs between the programmable logic regions. They have a fixed location on-board and consume a significant amount of logic around them. AMD/Xilinx UltraScale+ FPGAs [6] are also organized into multiple dies separated by silicon interposers. Crossing these die boundaries results in a much higher delay than on-die interconnect. Similar trends are also observed in case of Intel FPGAs [13]. Both AMD and Intel FPGA boards expose physical interfaces in the form of PCIe ports and networking interfaces in the form of Ethernet-compatible QSFP28 ports. Figure 2 describes the chip layout of the Alveo U55C, U250 cards and the Intel Stratix 10 cards.

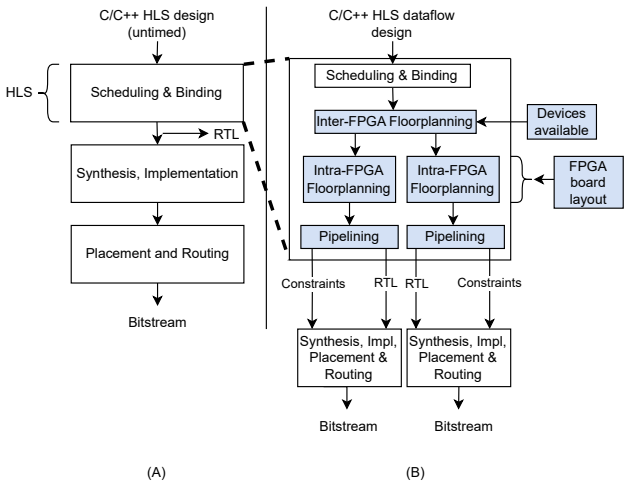

Figure 3. (A) Typical FPGA compilation flow, (B) Additions by TAPA-CS are highlighted in blue.

Accelerators for such FPGAs can be designed using commercial tools like Vitis [15] and Intel HLS [12]. Figure 3 (A) depicts the key steps involved in such toolflows. First, the untimed C/C++ input is converted into a timed RTL using HLS. Next, the timed RTL is passed for synthesis, placement and routing where the logic is mapped to the physical hardware. Despite enabling the increased use of FPGAs, these tools often suffer from poor quality of results compared with an expert-tuned RTL. This is because HLS tools cannot correctly estimate the final placement of compute modules on the board, and insert insufficient number of clock boundaries (registers) between them while converting the untimed input into a timed output. Several connections remain underpipelined, degrading the final frequency. Therefore, it is important to provide the tool a global view of the chip layout and the placement of the compute modules *during* HLS compilation.

Prior work such as Autobridge [38] integrates a coarse-grained floorplanning step with interconnect pipelining in HLS compilation for optimization on a single FPGA device. TAPA [27] extends HLS to feature fast compilation and user-friendly APIs which decouple communication and computation, allowing the user great flexibility in implementing the inter-module communication patterns. The new version of TAPA [37] integrates AutoBridge to improve design frequency, and we use this as one of the single FPGA baseline to evaluate TAPA-CS in Section 5.

3 Motivating Example of TAPA-CS

In this Section, we describe the importance of using TAPA-CS through the KNN example, and aim to dispel the common misconception that scale-out acceleration is only useful when the design cannot be routed on a single FPGA. We find that even when designs can be successfully routed on a

single device, span-out acceleration across multiple devices allows the design to efficiently utilize on-chip memory and HBM.

We use the KNN algorithm presented in [46]. The topology of the application is as illustrated in Figure 4 (A). There are two phases in this algorithm. The first phase calculates the distance of an input query data point with every other data-point in the dataset (blue modules). Given that the dataset contains N data points, each represented as a D-dimensional feature vector, this phase has a computational and memory access complexity of $O(N * D)$ for a single query. Since the KNN application is commonly used on very large datasets with large features, the cost quickly scales up. The second phase sorts the N distances calculated in phase 1 and returns the top K nearest neighbors (yellow modules). Since K is usually small and we only need to sort for the K smallest distances, the complexity of this phase is $O(N * K)$. Lastly, the green module in Figure 4 accumulates the final result and stores it back in HBM.

The design generated by traditional CAD tools results in a low design frequency of 165MHz and high latency. There are three main reasons for this. First, this design utilizes a port width and buffer size of 256 bits and 32KB which only saturates 51.2% of the HBM bandwidth. Prior work has also found that in case of memory-bound applications when multiple processing elements (PEs) access the HBM, the achievable bandwidth can drop to as low as 9.4GBps [29]. The optimal port width and buffer size which allows us to saturate the per-bank bandwidth is 512 bits and 128KB respectively. This configuration however, results in very high resource utilization in the lower die, leading to a failure in the routing phase of Vitis. Second, due to the smaller buffer, there is a higher number of HBM accesses which are 76x slower than on-chip memory accesses. Third, this design does not feature any floorplanning or interconnect pipelining. Using AutoBridge [38], the design frequency increases to 198MHz and still incurs a high latency since the HBM bandwidth cannot be saturated.

TAPA-CS includes all these considerations to generate an optimized KNN implementation automatically partitioned across two FPGAs as shown in Figure 4 (B). This provides sufficient resources in the lower die to route the optimal port width and buffer size configuration. Also, since the input data is divided between the two FPGAs, the compute load is balanced and neither FPGA is idle. The designs generated by TAPA-CS result in a design frequency of 300MHz and are 2.0x faster than designs on a single FPGA. Therefore, it is often a misconception that a multi-FPGA design is worse than a single FPGA design if it could be routed successfully on a single FPGA. Using multiple FPGAs can expose higher HBM bandwidth per compute module (aiding the performance of memory-bound applications), and enable the successful routing of larger designs (aiding the design of compute-bound applications). In Section 5, we scale this KNN design to 2-8 FPGAs by increasing the number of PEs.

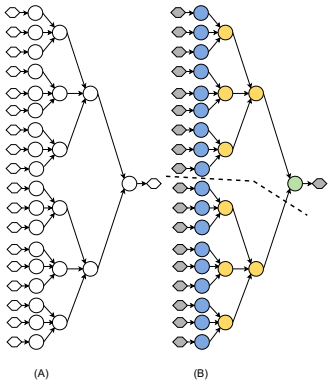

Figure 4. (A) Topology of the KNN application as found by the graph extraction step of TAPA-CS. Here, circles are compute modules (explained in Section 3), and hexagons indicate HBM access. (B) Partition found by TAPA-CS is indicated by the dashed line.

4 TAPA-CS Design

4.1 Problem Formulation

TAPA-CS takes as input a C/C++ dataflow program currently written in the TAPA-format [27] in which each function compiles into an RTL module and communicates with other functions using FIFOs. We also take the network topology and number of FPGAs present in the user cluster as input.

We model the input program as a graph $G(V, E)$, where each vertex ($v_i \in V$) is one of the functions, and the edges ($e_i \in E$) correspond to the FIFOs connecting them as shown in Figure 5 (A). Our goal is to map each vertex v_i to an FPGA F_i in the cluster such that the inter-FPGA communication cost is minimized while ensuring the compute-load between the multiple FPGAs is balanced. We explain how we model this cost in Section 4.3. We also apply several chip-level optimizations to ensure high frequency as discussed in Section 4.5.

4.2 Key Steps

There are seven major steps in TAPA-CS:

1. **Task graph construction**: We model the input workload as a graph $G(V, E)$ (Figure 5(A)).
2. **Task extraction and parallel synthesis**: We extract and synthesize each compute module in parallel to obtain an accurate resource utilization profile (Figure 5 (B)).
3. **Inter-FPGA floorplanning**: We use the resource utilization profile to intelligently floorplan this design across multiple FPGAs connected to each other through any topology (daisy-chained, ring, bus, star, mesh, hypercube, etc.) and data transfer protocol (PCIe,

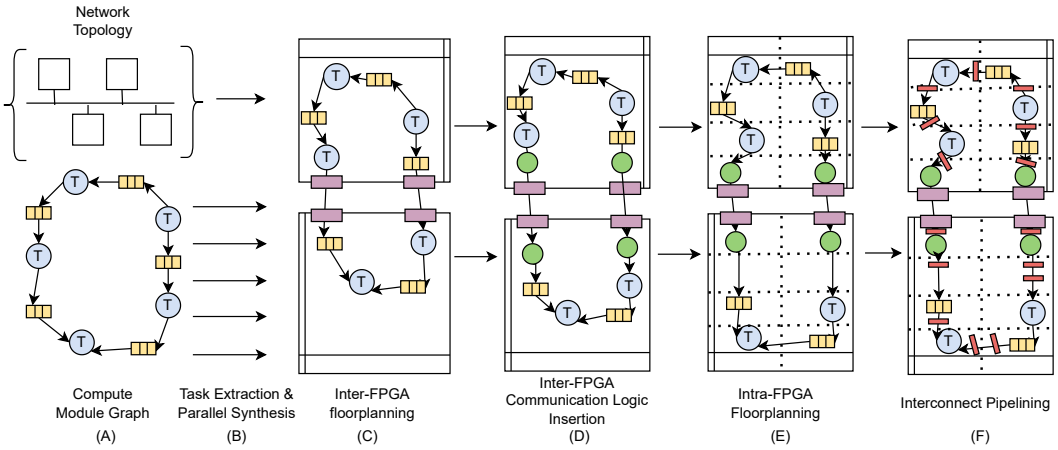

Figure 5. Key Steps in TAPA-CS

Ethernet, etc.). This step allows us to address the key limitation of traditional CAD toolflows (discussed in Section 2) by providing the scheduling and binding stage an accurate view of the topology and available programmable resources. We assign each compute module to an FPGA as shown in Figure 5 (C). We explain the details of this step in Section 4.3.

4. **Inter-FPGA communication logic insertion**: After mapping the design to multiple devices, we add the inter-FPGA communication logic (Figure 5(D)). We discuss details of the communication logic in Section 4.4.
5. **Intra-FPGA floorplanning**: We intelligently floorplan the design across each FPGA chip by providing the scheduling and binding stage information about the locations of the hard IPs, and I/O ports, and the internal chip layout (Figure 5 (E)). Section 4.5 details how we formalize this information and divide the FPGA into multiple slots.
6. **Interconnect Pipelining**: We add pipeline registers to the interconnect at the slot crossings to ensure high frequency designs (Figure 5 (F)). We also ensure correctness and that the final design execution cycles are not compromised by this interconnect pipelining step as discussed in Section 4.6.
7. **Bitstream generation**: Finally, the optimized designs and floorplanning constraints found by TAPA-CS are passed back into the traditional CAD stack to produce the bitstreams.

In the following Sections, we discuss each of these steps in detail and display the features through which our tool achieves high throughput and frequency accelerators.

4.3 Inter-FPGA Module Mapping

In this step, we provide the traditional CAD toolflows with a view of the available FPGA devices, their topologies, and

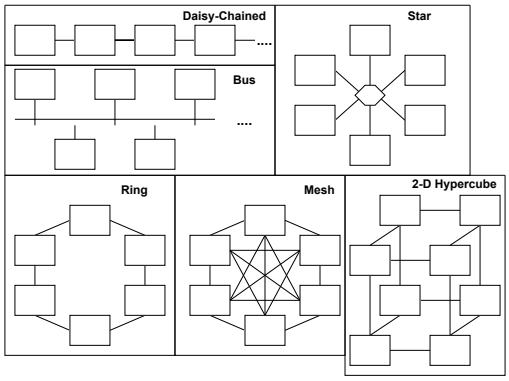

Figure 6. Network Topologies

the hierarchy of the user application, so that we can automatically find the optimal module-to-FPGA mapping. Design partitioning is enabled by the latency-insensitive nature of dataflow designs. Latency-insensitive design [23, 24] decouples the design of the interconnect from that of the compute modules. This allows the designer to add interconnect pipeline registers, and connections over different networks with arbitrarily long latency between compute modules, without affecting the functional correctness of the design.

In this mapping step, our goal is to minimize the high inter-FPGA communication cost while ensuring there is no resource congestion. If we are using two FPGAs, we consider the total available region to be divided into two grids as shown in Figure 5 (C).

Now, the placement task reduces to assigning each task to one of the two grids. For this, we use an ILP-based formulation to obtain exact partitioning solutions. While heuristic solvers are faster, ILP allows an accurate solution. We also show in Section 5 that both our intra- and inter- FPGA partitioning algorithms only add between 2.8-49.7 seconds

overhead to the overall compilation flow for number of compute modules ranging from 30 to 493, making the method scalable.

Let the binary variable denoting whether vertex v_i is placed on device F_i or not be v_d. Let the task $v_i \in V$ have a resource utilization profile of v_{area}. If r_v is the resources used by the set of tasks already placed in device F_i, then, before placing a new task v_i in this device, we need to ensure that there are enough resources. That is, for each type of on-chip resource,

$$\sum_{v \in r_v} v_d \times v_{area} < T, \tag{1}$$

where T is the threshold of utilization for each resource. Next, our ILP solver ensures that the cost of inter-FPGA communication is minimized. Therefore, we consider the placement of all the neighbors of task v_i in the cost function as follows:

$$\sum_{e_{ij} \in E} e_{ij}.width \times dist(F_i, F_j) \times \lambda \tag{2}$$

where $e_{ij}.width$ is the bitwidth of the FIFO channel connecting the two vertices v_i and v_j, function $dist(F_i, F_j)$ is a metric of the cost of communication between tasks v_i and v_j placed on the same or different FPGAs, and λ is a scaling factor to adjust the cost for different data transfer protocols like Ethernet and PCIe.

The communication cost function $dist(F_i, F_j)$ depends on the topology of the network-connected FPGAs. Consider the network topologies shown in Figure 6. In case the FPGAs are daisy-chained,

$$dist(F_i, F_j) = |F_i.device\_num - F_j.device\_num| \tag{3}$$

where $F_i.device\_num$ and $F_j.device\_num$ are the device IDs associated with the FPGAs. Similarly, in the case of a bidirectional ring topology, the distance metric changes to:

$$dist(F_i, F_j) = \min(|F_i.device\_num - F_j.device\_num|, \\ (total\_num - |F_i.device\_num - F_j.device\_num|))$$

where $total\_num$ is the total number of FPGAs in the ring.

The scaling factor λ is used to adjust the cost in a system with multiple interconnection media. We use Ethernet-based connections offering 100GBps bandwidths as the baseline and scale the cost for other media accordingly. For example, if the interconnection used is PCIe Gen3x16, then the cost is scaled by a factor of 12.5 compared with the cost of using Ethernet-based connections.

Note that our partitioner does not always recommend the min-cut. For the placement of each module, we consider the resource and communication cost added by its placement on- and off-chip. In case the module can be accommodated on the same chip as its neighbors, we would pay a high price by moving the compute module off-chip than placing it on-chip. However, if the placement of this module on-chip results in congestion (which in turn lowers design frequency), it would be more beneficial to place the module off-chip even at the cost of increased inter-FPGA connections. This trade-off ensures that we can achieve high frequency designs on both FPGAs.

4.4 Inter-FPGA Communication

Despite the excellent opportunity to leverage networking capabilities in modern FPGAs, existing CAD tools do not explicitly support networking. TAPA-CS supports a library of inter-FPGA communication protocols, such as Ethernet-based RoCEv2, and PCIe-based P2P DMA [16]. However, for the scope of this paper, we limit our discussions and evaluations to using the QSFP28 Ethernet ports. Here, we use AlveoLink [3] (Figure 7) to add networking support to the existing toolflows. AlveoLink ensures reliable, lossless, and in-order data transfer with a low resource overhead of ~5% on the Alveo U55C cards. RoCEv2 implements priority flow control to ensure that low priority traffic is paused, allowing AlveoLink traffic to continue without loss in case of congestion. It offers a low round-trip data transfer latency of 1 μs between two FPGAs and is 12.5x faster compared with PCIe Gen3x16 -based connections. AlveoLink's main components are as follows:

1. HiveNet IP: This is a Vitis-compatible implementation of the RoCE v2 protocol which directly connects to user kernels.
2. CMAC kernel: This is a board-specific interface between the signals detected at the QSFP28 ports and the signals detected at the commodity network.

The user logic and AlveoLink IPs (free-running kernels [9]) are connected through streaming connections in the connectivity config file [8], which inserts a FIFO between them. The FIFOs are implemented using on-chip memory (BRAM). Free running kernels do not need to be explicitly started, and operate on data as soon as it is available either on the QSFP28 ports or from the user logic. Therefore, in case of long pauses between the source and destination FPGAs, the kernels simply stall until the required data is received. Each port of each peer in the network is equipped with one HiveNet and CMAC IP. At startup, the host assigns each IP a unique ID using the card's IP and MAC subnet. When the user logic needs to write a packet off-chip, the payload is appended with the destination peer's assigned ID. Note that the user logic can route data to multiple peers based on the destination address specified with the payload. Based on the topology generated in Section 4.3, TAPA-CS automatically generates the code required to read/write the in-/out-bound packets (along with the meta-data of the source/destination) from/to the FIFO connecting the user logic and the AlveoLink IP. AlveoLink supports up to 8192 lossless connections, each of which require an independent send/receive queue pair. We discuss limitations of AlveoLink in Sections 5 and 7.

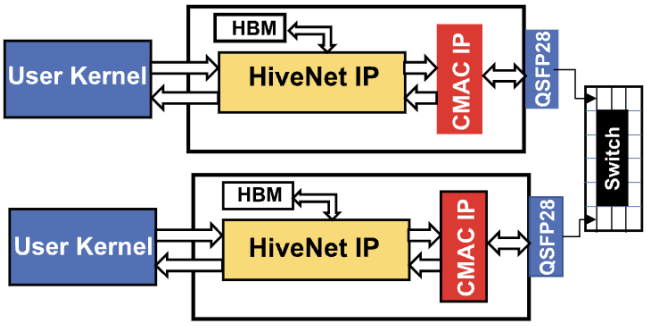

Figure 7. AlveoLink features as described in [3]

4.5 Intra-FPGA Module Mapping

After assigning each task to the set of devices in our cluster and adding the inter-FPGA communication interfaces, the next step is to apply a similar top-down partitioning approach to each FPGA (Figure 5(E)). To formalize the device-specific information and present it to the scheduling and binding stage, we view each FPGA as a grid divided into slots by the hard IPs and static regions. For example, the Alveo U55C card shown in Figure 2 is presented to TAPA-CS as a grid with 6 slots divided into two columns and 3 rows. Our goal is to place each vertex in one of these slots based on the resource utilization ratios per slot and the cost of connecting this module to all its neighbors. In this step, our goal is to minimize the cost of inter-die communication. Therefore, the new cost function is:

$$\sum_{e_{ij} \in E} e_{ij}.width \times (|v_i.row - v_j.row| + |v_i.col - v_j.col|) \quad (4)$$

where $v_i.row$, $v_j.row$ represent the rows in which tasks v_i and v_j are placed respectively, and the same for the columns. We continue such a two-way ILP-based partitioning scheme until we divide each FPGA into eight grids.

One of the key considerations in this step is the optimal usage of the HBM channels. Consider the example of the U55C cards. While the HBM offers a high aggregate bandwidth of 460GBps, it is still 76x slower than on-chip data accesses. Therefore, it is important to utilize the HBM channels exposed to the user kernel optimally. As shown earlier in Figure 2, all the HBM channels on-board the U55C are exposed in the bottom-most die. Suboptimal HBM channel binding can result in large routing delays and increase congestion in this die, leading to routing failure. Therefore, TAPA-CS supports an automatic HBM channel binding exploration where we find the optimal mappings based on the workload characteristics.

4.6 Interconnect Pipelining

Following the intra-FPGA optimizations, we also conservatively pipeline the interconnect at all slot-crossings to prevent long delays from degrading the final clock frequency. In contrast to prior work like [24, 43], we conservatively pipeline all slot-crossing wires because each of our compute modules compiles into an RTL controlled by a finite state machine (FSM). Therefore, it is difficult to estimate the latency added by the pipelining step. Next, to ensure that the design throughput is not negatively affected by the additional pipeline registers, we also balance the latency of parallel paths based on cut-set pipelining [50] as shown in [38]. In this step, the latency added by reconvergent paths is balanced to ensure that final correctness is not impacted. The pipeline FIFOs are indicated in red in Figure 5(F).

5 Evaluation

We implement TAPA-CS in Python and integrate it with Vitis 2022.1. Either MIP [14] or the Gurobi solver [10] (free for academia) can be used to solve our ILP formulations described in Sections 4.3 and 4.5. We test TAPA-CS on a server equipped with two nodes, each featuring four Xilinx Alveo U55C cards connected through their QSFP28 ports in a ring topology. The server features a 128 core AMD EPYC 7V13 CPU operating at 2.45GHz.

5.1 Benchmarks and Baselines

We evaluate TAPA-CS over multiple variations of the following benchmarks (topology shown in Figure 8):

1. Stencil Dilate: This is a 2-D 13-point stencil kernel from the Rodinia HLS benchmark [31, 58] which is used in biomedical research to track leukocytes in blood vessels. We test this kernel over 64 to 512 iterations.
2. Page Rank created by [27]: This kernel features four PEs and one central controller with dependency cycles between the compute modules. It implements the algorithm described in [49]. We test this design over multiple networks taken from the SNAP dataset collection [45].
3. KNN created by [46]: This kernel contains 17 compute modules implementing an optimized accelerator for calculating each data point's distance to its neighbor, and sorting the distances to obtain the K-nearest neighbors. We test this design across varying input sizes and feature dimensions.
4. Systolic-array CNN accelerators created by AutoSA [60]. This systolic array accelerator consists of multiple PEs arranged in a grid format, with a total of 493 compute modules. The CNN we choose is an implementation of the third layer of the VGG model [54]. We test TAPA-CS on multiple grid dimensions ranging from 13 x 4 to 13 x 20.

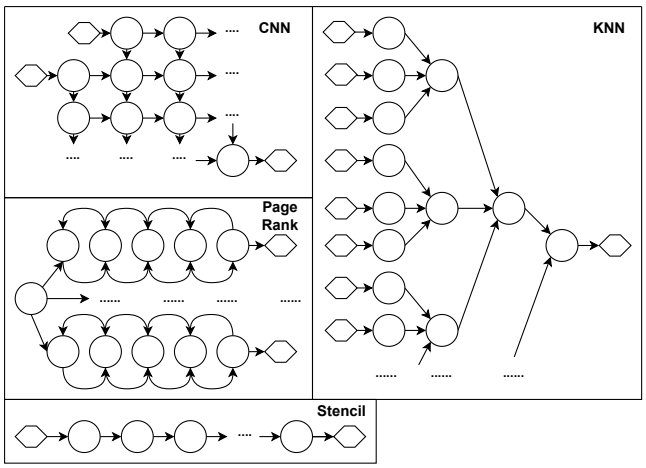

Figure 8. Topology of benchmarks. Here, circles represent compute modules while hexagons represent HBM access.

We compare the latency and frequency of TAPA-CS with two baselines, namely F1-V and F1-T. Here, F1-V is a single FPGA implementation generated through Vitis HLS and F1-T is a single FPGA implementation generated through TAPA/AutoBridge [37]. The TAPA-CS designs are denoted by F2 (2 FPGAs), F3 (3 FPGAs), and F4 (4 FPGAs).

Table 2 summarizes the speed-up obtained by TAPA-CS for each benchmark averaged across all the datasets and configurations tested. Note that the speed-ups are normalized with respect to the single FPGA design generated by Vitis HLS. We discuss more details of each application in the following Sections and also discuss the reasons for the varying speed-ups.

We discuss the floorplanning and resource overheads added by TAPA-CS in Section 5.6. Lastly, we evaluate designs spanning across multiple nodes and 8 FPGAs in Section 5.7.

Benchmark	F1-V	F1-T	**F2**	**F3**	**F4**
Stencil	1	1.25×	**1.71×**	**2.37×**	**3.06×**
PageRank	1	1.54×	**2.64×**	**4.28×**	**5.98×**
KNN	1	1.2×	**1.72×**	**2.53×**	**3.60×**
CNN	1	1.1×	**1.41×**	**2.0×**	**2.54×**

Table 2. Speed-up of TAPA (F1-T), and TAPA-CS (F2, F3, and F4) normalized against the Vitis HLS (F1-V) baseline.

5.2 Stencil

Stencil kernels apply a sliding window (or stencil) of computation over an input array to produce an output array. Such kernels can either be memory-bound or compute-bound based on the input size and the number of iterations. Prior work [58] found that for a fixed input size, stencil designs with a smaller number of iterations are memory-bound while designs with a larger number of iterations are compute-bound. We chose the Dilate kernel from the Rodinia HLS benchmark to test TAPA-CS. It is a 2D 13-point kernel which we test for an input size of 4096×4096 and iterations varying between 64 and 512.

Iters	Ops/Byte	Volume (MB)
64	208	144.22
128	416	288.43
256	832	576.86
512	1664	1153.73

Table 3. Stencil: Compute intensity (operations / byte of external memory access) and total inter-FPGA data transfer over varying iterations and a fixed input size of 4096×4096.

The single FPGA baseline design can successfully route a design with 64 iterations, 15 PEs, and an HBM access bit-width of 128. A single device cannot support a larger number of PEs, iterations, and bit-widths. We measure the compute intensity (in terms of operations per byte of external memory access) and total inter-FPGA data transfer volume of various iteration configurations in Table 3. Note that the compute intensity calculations assume optimal data reuse. We find that iterations 64 and 128 are memory-bound while iterations 256 and 512 are compute-bound. Therefore, we scale the design to suit the additional on-chip resources and HBM bandwidth available in the multi-FPGA scenario as follows:

1. Design with 64 and 128 iterations: Increase HBM access bit-width from 128 to 512 as well as the channels used by the design from 32 (single FPGA) to 64 (2 FPGAs), 96 (3 FPGAs), and 128 (4 FPGAs).
2. Design with 256 and 512 iterations: Increase the number of PEs from 15 (single FPGA), to 30 (2 FPGAs), 60 (3 FPGAs), and 90 (4 FPGAs) keeping the HBM access bit-width at 128.

The overall runtime of the different configurations tested on up to 4 FPGAs is shown in Figure 9. For the design with 64 iterations, the 4 FPGA design is 4.9× faster than the single FPGA design produced through Vitis. The main reasons for this are the increased HBM channel usage and the higher bit-width. However as the number of iterations increases, the relative gains obtained by using multiple FPGAs decreases. The 4 FPGA design with 512 iterations is only 2.3× faster than the Vitis baseline. This is because in the case of the 4 FPGA design with 64 iterations, each FPGA has a compute-intensity of 52 ops/byte with an inter-FPGA transfer size of 144.22MB. However, for the design with 512 iterations, each FPGA has a compute intensity of 416 ops/byte with an inter-FPGA transfer size of 1.1GB. Large inter-FPGA transfer sizes lead to higher idle PE time. Also, unlike other applications we discuss in Sections 5.3 and 5.4 (ie., PageRank and KNN), in the case of the stencil application, each FPGA operates sequentially due to the topology of the design. That is, FPGA

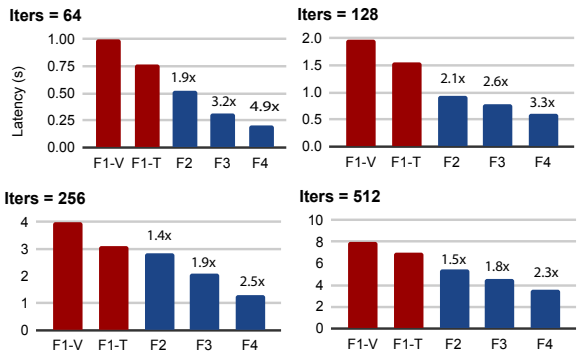

Figure 9. Stencil: Latency comparison between TAPA-CS (F2, F3, F4), TAPA (F1-T), Vitis-HLS (F1-V).

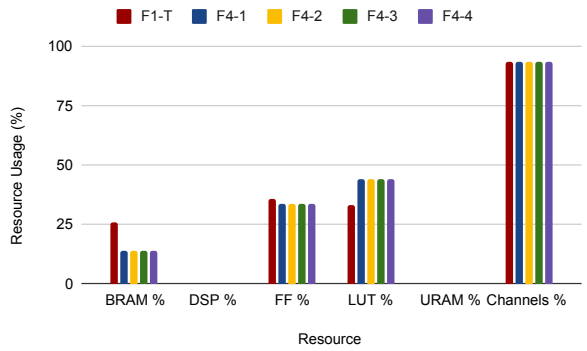

Figure 10. Stencil: Resource utilization of single FPGA baseline (F1-T) and 4-FPGA design (F4). Here, F4-1 to F4-4 denote the 4 FPGAs used in design F4.

2, 3, and 4 lie idle while their predecessor executes. However, as the number of iterations increase and the inter-FPGA transfer sizes rise, this becomes a limitation.

Figure 10 displays the resource utilization of the single FPGA baseline and each of the 4 FPGAs used in design F4. Since the resource utilization profiles per FPGA in case of F2 and F3 are similar to that of F4, we do not report them.

The single FPGA baseline achieves a design frequency of 165 MHz using Vitis, 250MHz using TAPA, and each of the 2, 3, and 4 FPGA designs generated by TAPA-CS achieve 300MHz, displaying a 81.8% increase compared to Vitis, and a 20% increase compared with TAPA.

5.3 Page Rank

The topology of the Page Rank application is illustrated in Figure 8. This accelerator implements the citation ranking algorithm described in [27, 49]. First the input graph is pre-processed on the host and loaded onto the device HBM. Then, the edges are streamed to each PE on-chip, which calculate and propagate weighted rankings from source vertex to destination vertex. These updates are stored back into the HBM before they are accumulated over each vertex to calculate the final ranking. The algorithm runs until convergence and uses an edge-centric model such that edges are traversed to avoid bank conflicts, and not by either source or destination vertices.

The single FPGA baseline can successfully route a design with 4 PEs, using 27 HBM channels to read the edges, and store the intermediate data. We scale the design to increase the number of PEs from 4 (single FPGA), 8 (2 FPGAs), 12 (3 FPGAs), and to 16 (4 FPGAs). Note that in the case of the PageRank application, the inter-FPGA transfer volumes depend on the dataset used and do not change with the number of PEs in the design. Therefore, as the compute intensity of the application scales with the increase in PEs, the inter-FPGA transfer size remains constant for a given dataset. This is in contrast with the Stencil application discussed earlier where the transfer volumes depend on the iterations and not the input size. Also, based on the topology depicted in Figure 8, once the FPGA containing the leftmost module (FPGA 1) routing data from the HBM to the different PEs is executed, other FPGAs (FPGA 2-4) can be launched in parallel. These two factors aid in the scalability of the PageRank application.

We test the design on five networks of varying number of nodes and edges taken from the SNAP dataset [45] shown in Table 4.

The overall runtime of the different PE configurations tested on up to 4 FPGAs is shown in Figure 11. Considering that the data transfer volumes do not increase with the increase in compute intensity for this application, each of the tested configurations benefit from scaling, leading to superlinear speed-ups. The 2, 3, and 4 FPGA designs are on average 2.64×, 4.28×, and 5.97× faster than the single FPGA Vitis baseline across all the tested datasets.

Network	Nodes	Edges
web-BerkStan	685,230	7,600,595
soc-Slashdot0811	77,360	905,468
web-Google	875,713	5,105,039
cit-Patents	3,774,768	16,518,948
web-NotreDame	325,729	1,497,134

Table 4. Networks used to test PageRank.

Figure 12 displays the resource utilization of the single FPGA baseline and each of the 4 FPGAs used in design F4. The resource utilization profiles per FPGA in case of F2 and F3 are similar to that of F4, therefore, we do not report them.

The single FPGA baseline achieves a design frequency of 123MHz using Vitis, 190MHz using TAPA, and each of the 2, 3, and 4 FPGA designs generated by TAPA-CS achieve 266 MHz, displaying a 116.26% increase compared to Vitis, and 40% increase compared with TAPA.

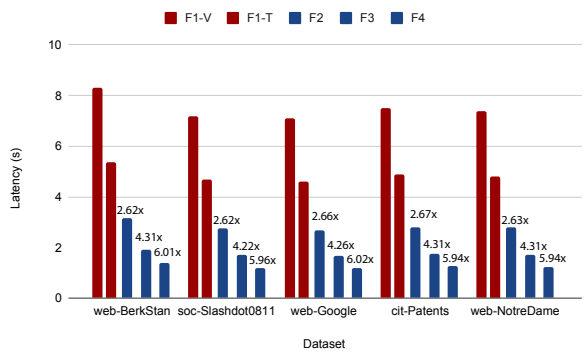

Figure 11. PageRank: Latency comparison between TAPA-CS (F2, F3, F4), TAPA (F1-T), Vitis-HLS (F1-V).

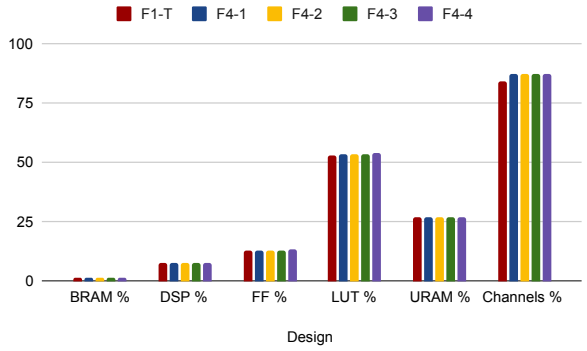

Figure 12. PageRank: Resource utilization of the single FPGA baseline (F1-T) and 4-FPGA design (F4). Here, F4-1 to F4-4 denote the 4 FPGAs used in design F4.

5.4 KNN

We use the KNN accelerator designed by [46], which demonstrates quadratic growth in memory access and computational complexity as discussed in Section 3. The single FPGA baseline can successfully route a design with a port width and buffer size of 256 bits and 32KB with 27 compute modules (Figure 4). Note that the scale of the design is limited by the HBM ports available to the blue compute modules for reading the input data and calculating the pairwise distances. Therefore, we scale the design such that the 2-4 FPGA cases use 36, 54, and 72 blue modules reading from the HBM. The rest of the modules (yellow and green) responsible for sorting the distances and accumulating the final result are also scaled accordingly. To evaluate the effectiveness of the application partitions found by TAPA-CS, we test across varying dataset sizes (N) and dimensions (D) as shown in Table 5. The total size of the search space ($N * D * sizeof(float)$) varies from 8MB (N=1M, D=2) to 4GB (N=8M, D=128).

Note that based on the partition found in Figure 4, the size of the inter-FPGA data transfer only depends on the value of K, and is independent over the varying search space. Therefore, unlike in the case of the Stencil application discussed earlier, as the computational and memory access complexity of the KNN application scales, the size of the data transfer between FPGAs remains constant. Another difference is that in the case of the KNN application, all FPGAs except the last FPGA (responsible for aggregating the final data using the green module), can be launched completely independently, and do not require any data from each other.

Figure 13 presents the speed-up of the design generated by TAPA for a single FPGA and TAPA-CS over 2-4 FPGAs compared with the Vitis HLS baseline over varying feature dimensions. The 2, 3, and 4 FPGA designs produced by TAPA-CS are on average 2×, 2.7×, and 3.9× faster than the baseline Vitis version. Next, we vary the dataset sizes and evaluate the performance of TAPA-CS over 2-4 FPGAs in Figure 14. Compared with the Vitis-generated single-FPGA baseline, the 2, 3, and 4 FPGA designs are on average 1.7×, 2.8×, and 3.9× faster. Compared with the TAPA-generated single FPGA baseline, the designs are 1.4×, 2.3×, and 3.2× faster.

Parameters	Values
N: Number of data points in the dataset	1M, 2M, 3M, 4M, 8M
D: Dimension of the feature vectors	2, 4, 8, 16, 32, 64, 128
K	10

Table 5. KNN parameters

Figure 15 displays the resource utilization of the single FPGA baseline and each of the 4 FPGAs used in design F4. The resource utilization profiles of designs F2 and F3 are similar to that of F4 per FPGA. Therefore, we do not report them. The single FPGA baseline achieves a design frequency of 165MHz using Vitis, 198MHz using TAPA, and each of the 2, 3, and 4 FPGA designs generated by TAPA-CS achieve a design frequency of 220MHz displaying a 33% increase compared with Vitis and 11.11% increase compared with TAPA.

5.5 CNN

The CNN accelerator we chose is a systolic-array based implementation of the third layer of the VGG model [54] generated by AutoSA [60]. This accelerator consists of a configurable grid of PEs performing identical computations.

The single FPGA baseline can successfully route a design with a grid size of 13x4 using Vitis and 13×8 using TAPA. Larger designs like 13×12, 13×16, and 13×20 either cause congestion or require more resources than what is available on a single device. We display the resource utilization profiles of these configurations in Table 7. Table 6 presents the inter-FPGA transfer volumes for the different grid configurations. In case of the CNN application, the compute intensity depends on the input size (54.5M floating-point

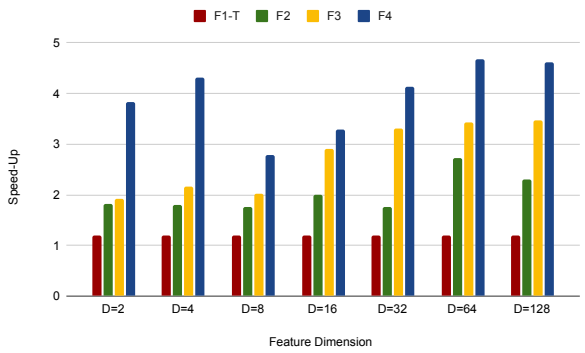

Figure 13. KNN: Speed-up of TAPA (F1-T), TAPA-CS (F2, F3, F4) compared with Vitis HLS baseline over varying feature size (K=10, N=4M).

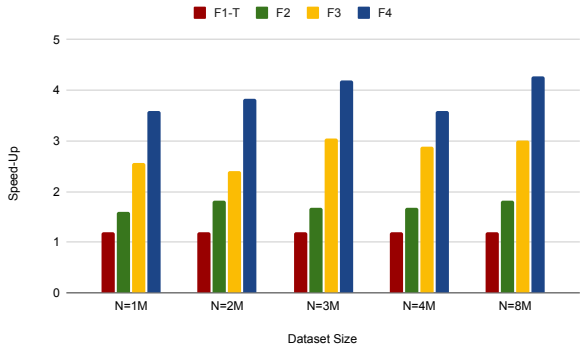

Figure 14. KNN: Speed-up of TAPA (F1-T), TAPA-CS (F2, F3, F4) compared with Vitis HLS baseline over varying dataset size (K=10, D=2).

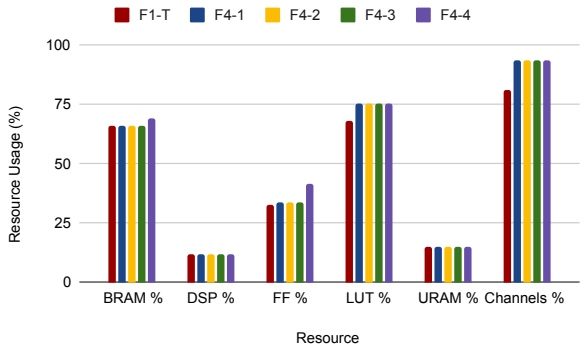

Figure 15. KNN: Resource utilization of single FPGA baseline (F1-T) and 4-FPGA design (F4). Here, F4-1 to F4-4 denote the 4 FPGAs used in design F4.

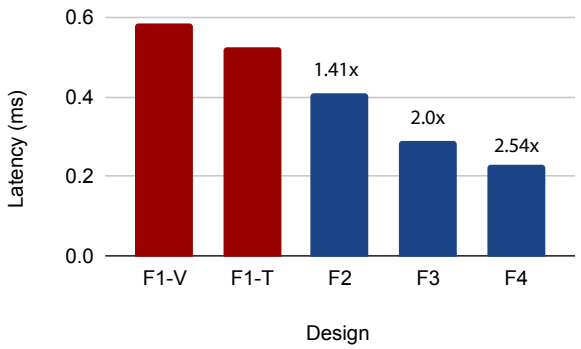

Figure 16. CNN: Latency comparison between TAPA-CS (F2, F3, F4), TAPA (F1-T), and Vitis-HLS (F1-V).

operations), and the inter-FPGA transfer volumes increase with the increase in grid size.

We evaluate the single FPGA baselines (13×4 through Vitis and 13×8 through TAPA) against a 2 FPGA design with the 13×12 grid, a 3 FPGA design with the 13×16 grid, and a 4 FPGA design with the 13×20 grid in Figure 16. Compared with the single FPGA Vitis baseline (13×4), the 2-FPGA design (13×12) is 1.41× faster, the 3-FPGA design (13×16) is 2.0× faster, and the 4-FPGA design (13×20) is 2.54× faster. There are two main reasons limiting the speed-up of CNNs when we scale to multiple FPGAs. First, as the grid size increases, the inter-FPGA data transfer sizes also increase (Table 6). Second, due to the grid-like structure of the systolic-array, there are more PEs writing to AlveoLink requesting inter-FPGA data transfer than in the case of the other applications shown in Figure 8. Due to the contention, there is a higher occurance of idle PEs on the second FPGA.

Grid Size	Volume (MB)
13×4	2.14
13×8	4.28
13×12	6.42
13×16	8.57
13×20	10.71

Table 6. Inter-FPGA data transfer volumes over grid sizes.

The single FPGA baselines achieve a design frequency of 300MHz for 13×4 using Vitis, and 13×8 using TAPA, but they fail placement and routing for the larger grid sizes. TAPA-CS can successfully route grid sizes 13×12, 13×16, and 13×20 on 2, 3, and 4 FPGAs achieving a design frequency of 300MHz on each FPGA.

5.6 Overheads Added by TAPA-CS

TAPA-CS adds the following overheads to the overall HLS compilation stage and the resource utilization per FPGA:

Grid	LUT %	FF%	BRAM%	DSP%	URAM%
13×4	20.4	12.1	14.2	25.2	0
13×8	38.3	23.5	23.7	49	0
13×12	56.1	34.3	32.7	80.1	0
13×16	74	45.7	42.3	97.6	0
13×20	91.9	57	52.1	123.7	0

Table 7. CNN: Resource utilization of different grid sizes.

1. Floorplanning overheads: To test the scalability of TAPA-CS, we study the floorplanning overheads added by the tool to the HLS compilation stage for the CNN benchmark. This is the largest application tested by TAPA-CS with 493 compute modules and 925 FIFO connections. The inter- and intra-FPGA floorplanning steps add an overhead between 0.3s-24.6s and 0.1s-12.9s respectively for the different grid sizes described in Table 7 using the Gurobi solver.
2. Resource overheads: The networking IPs add the following negligible resource overhead per QSFP28 port per board: (1) LUT: 2.04%, (2) FF: 2.94%, (3) BRAM: 2.06%, (4) DSP: 0%, and (5) URAM: 0%.

5.7 Scalability Beyond a Single Server Node

The experimental setup used to evaluate TAPA-CS consists of two nodes with 4-FPGA rings. To scale a design beyond a single node, we utilize host-side MPI and a 10Gbps ethernet link to transfer intermediate data. Table 8 describes the hierarchy of bandwidths involved in multi-FPGA design. We evaluate such an 8 FPGA setup for the following two applications:

1. Stencil: We use the design with 512 iterations and scale the number of PEs to 120. As discussed in Table 3, each FPGA transfers a total data volume of 1.1GB to the next FPGA. The overall runtime of the 8-FPGA setup is 11.65s, which is 1.45× slower than the single FPGA Vitis baseline. In this case, the main bottleneck is the high data movement, and sequential nature of the stencil application which leaves most of the FPGAs idle. The intermediate data from the first set of the 4-FPGA ring first needs to be transferred from the device memory to the host memory. Next, the data must be moved from one node to the other using the 10Gbps link. Lastly, the data has to be moved from the host memory of the second node to the device memory of the second 4-FPGA ring.
2. PageRank: We scale the PageRank application from 4 PEs (single FPGA) to 32 PEs for the 8-FPGA design and test it on the cit-Patents dataset described in Table 4. Note that unlike in the case of the Stencil application where each FPGA executes sequentially, in the case of PageRank (Figure 8), once the vertex router module completes execution on the first FPGA, all other FPGAs can be launched in parallel, increasing the scalability of the design. The 8-FPGA design achieves an end-to-end latency of 3.44s, which is 1.4× faster than the single FPGA Vitis baseline. Despite the speed-up obtained, the 8-FPGA design is still slower than the 2-FPGA design on a single node. Therefore, the inter-node network adds significant latency, reducing the benefits of scaling.

Transfer	Bandwidth
On-chip (SRAM)	35TBps
Off-Chip (HBM)	460GBps
Inter-FPGA	100Gbps
Inter-Node	10Gbps

Table 8. Hierarchy of data transfer bandwidths involved in multi-FPGA design.

6 Related Works

6.1 Prior Work on Inter-FPGA Communication

Prior attempts at leveraging the networking capabilities exposed by modern FPGAs can be classified into two categories. The first category of work uses host-orchestrated data transfers [40, 52, 53]. In these works the host coordinates the inter-accelerator communication by exposing programmer-friendly MPI-like primitives. Using host-orchestration avoids re-programming the FPGA bitstream whenever a different communication pattern is to be followed. However, considering that several dataflow FPGA workloads suit streaming, where data is produced and consumed every cycle, host-orchestration adds significant overheads. The second category of works uses device-side initiation of inter-FPGA communication [32, 39, 56].

We compare prior work and AlveoLink (described in Section 4.4 in Table 9. Compared with EasyNet [39] which also achieves a similar data transfer throughput of 90GBps, AlveoLink requires about half of the on-board resources.

Project	Orchestration	Resource (%)	Performance (GBps)
TMD-MPI[53]	Host	26	10
Galapagos[56]	Device	11.5	10
SMI[32]	Device	2	40
EasyNet[39]	Device	10	90
ZRLMPI[52]	Host	-	10
ACCL[40]	Host	16	80
AlveoLink[3]	**Device**	**5**	**90**

Table 9. Prior work addressing Challenge 1. Here, "-" implies that the area overhead is unknown. AlveoLink (used by TAPA-CS) is indicated in bold.

6.2 Prior Work on Partitioning, Mapping, & High Design Frequency

There are some initial research efforts which address partitioning and mapping across multiple FPGAs which provide a good starting point for TAPA-CS, but they suffer from several shortcomings. Elastic-DF [17] propose an ILP-based partitioner similar to ours, but suffer from poor frequency (190-240MHz) as they do not couple floorplanning and interconnect pipelining with HLS compilation. Also, Elastic-DF is integrated with the DNN inference compiler FINN [21, 59], leading to poor generalizability to different workloads.

A different approach to design partitioning leverages latency-insensitivity [23, 24] to break the design at the latency-insensitive endpoints [34, 61, 62]. [34] leverage this technique, but expect the user to provide a static module-to-FPGA mapping, use Verilog/VHDL, and perform simulation-based experiments. In case of TAPA-CS, we automatically find the optimal module-to-FPGA mapping, expose a C++ interface to the user, and perform experiments on real FPGAs achieving high frequency. Virtualization-based works such as [61–63] assign modules to pre-placed and pre-routed "soft blocks" which does not scale well to real-world large-scale designs where each function is compiled into an RTL controlled by a finite state machine (FSM).

SMAPPIC [28] introduces a multi-node emulation system where each node can be a single die of an FPGA or the whole FPGA. It uses the computational cores shipped with BYOC [18] assign cores to the nodes. However, SMAPPIC uses Gen3x16 PCIe-based connections between FPGAs which provides a slow round-trip latency of 1250ns, and does not provide the tool with a hardware layout of the FPGAs. This leads the designs to have a low final frequency of 100MHz. In contrast, TAPA-CS has an inter-FPGA round trip latency of 1 μs (12.5x faster than SMAPPIC), provides the partitioning tool with a global view of the chip layout allowing us to achieve a high frequency of between 266-300MHz. Also, in TAPA-CS a single die can contain any number of modules, and modules spanning across multiple dies are pipelined sufficiently to maintain the final frequency.

7 Challenges in Multi-FPGA Design

Modern FPGAs are well-equipped to support accelerator designs spanning across multiple devices and nodes as displayed in Section 5. However, compared with the efforts in design across CPUs and GPUs, the FPGA community still has several open challenges to address. We outline some of them below:

1. Workload perspective: Most FPGA designers rely on domain-specific knowledge to produce accelerators optimized to the resources available on a single device. However, with network-connected multi-FPGA environments, FPGA designers need to think beyond a single FPGA and scale their designs accordingly. CPU-based designs can be scaled easily through multi-threading and GPU-based designs offer data-parallel and model-parallel modes through the PyTorch front-end which make it easier for users to design large applications without considering the physical constraints of each device. TAPA-CS provides a partitioning framework for large scaled-up designs. However, there is a lack of frameworks which automatically enable scaling-up a design from a single FPGA to multiple FPGAs. We are currently working on map-reduce style programming frameworks for FPGAs which will allow automated scaling based on the memory-/compute-intensity of the application, combined with the partitioning introduced in this paper.
2. Inter-FPGA communication perspective: As discussed in Section 6, AlveoLink offers the best theoretical latency (1μs) at the lowest resource budget. However, in practice, the latency varies greatly with different packet sizes and total volume of data. For example, a data transfer of 64MB with packet size of 64B takes a total of 6.5ms, while the same volume with a packet size of 128B takes a total of 3.9ms. This overhead can be significant for applications with strict latency constraints. Ideally, FPGA-based communication should add an overhead in the order of <100ns to be widely adopted for different applications. We are working with FPGA vendors to collaborate on such a solution.

8 Conclusion

This paper presents TAPA-CS - a task-parallel dataflow programming framework which automatically partitions, generates, and compiles a large design across a cluster of FPGAs achieving high throughput and frequency. TAPA-CS uses an ILP-based partitioning framework which takes into account resource utilization profiles of compute modules before mapping them to different FPGAs. It also couples a coarse-grained floorplanning step with pipelining at the inter- and intra-FPGA levels to ensure high accuracy. We test the design across multiple benchmarks of varying compute and memory requirements to validate the scalability and generalizability of the tool. Across all the tested designs, TAPA-CS achieves an average throughput improvement of 2.1×, 3.2×, and 4.4× using 2, 3, and 4 FPGAs respectively, and frequency improvement between 11-116% compared with single FPGA designs generated through Vitis HLS.

9 Acknowledgments

This work is partially supported by the PRISM Center under the JUMP 2.0 Program, the NSF RTML Award CCF-1937599, the CDSC industry partners, and AMD HACC Program. The authors would like to thank AMD for providing the multi-FPGA testbeds and support for AlveoLink. J. Cong has a financial interest in AMD.

References

[1] Alibaba FPGAs in the Cloud. https://www.alibabacloud.com/help/en/fpga-based-ecs-instance.

[2] Alveo U55C High Performance Compute Card. https://www.xilinx.com/products/boards-and-kits/alveo/u55c.html#specifications.

[3] AlveoLink. https://github.com/Xilinx/AlveoLink.

[4] Amazon AQUA. https://aws.amazon.com/redshift/features/.

[5] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[6] AMD/Xilinx UltraScale+ Devices Overview. https://docs.xilinx.com/r/en-US/ug1120-alveo-platforms/Overview.

[7] Baidu FPGAs in the Cloud. https://intl.cloud.baidu.com/product/bcc.html.

[8] Connectivity Options. https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration/connectivity-Options.

[9] Free Running Kernels in Vitis HLS. https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration/Free-Running-Kernels.

[10] Gurobi Solver. https://www.gurobi.com/downloads/gurobi-optimizer-eula/.

[11] Huawei FPGAs in the Cloud.

[12] Intel HLS. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/hls-production-brief.pdf.

[13] Intel Stratix 10.

[14] Python MIP. https://www.python-mip.com/.

[15] Vitis HLS 2022.2. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.

[16] Xilinx PCIe-Based P2P. https://xilinx.github.io/XRT/master/html/p2p.html.

[17] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning. *ACM Trans. Reconfigurable Technol. Syst.*, 15(2), dec 2021.

[18] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 699–714, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Chaim Baskin, Natan Liss, Evgenii Zheltonozhskii, Alex M. Bronstein, and Avi Mendelson. Streaming Architecture for Large-Scale Quantized Neural Networks on an FPGA-Based Dataflow Platform. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, may 2018.

[20] Chaim Baskin, Natan Liss, Evgenii Zheltonozhskii, Alex M. Bronstein, and Avi Mendelson. Streaming Architecture for Large-Scale Quantized Neural Networks on an FPGA-Based Dataflow Platform. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 162–169, 2018.

[21] Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks, 2018.

[22] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, may 2011.

[23] L.P. Carloni, K.L. McMillan, A. Saldanha, and A.L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, pages 309–315, 1999.

[24] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[25] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[26] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 204–213, 2021.

[27] Yuze Chi, Licheng Guo, Jason Lau, Young kyu Choi, Jie Wang, and Jason Cong. Extending High-Level Synthesis for Task-Parallel Programs, 2021.

[28] Grigory Chirkov and David Wentzlaff. SMAPPIC: Scalable Multi-FPGA Architecture Prototype Platform in the Cloud. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 733–746, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 116–126, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38:8–20, March 2018.

[31] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, 2018.

[32] Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, and Torsten Hoefler. Streaming Message Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[33] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[34] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. Leveraging Latency-Insensitivity to Ease Multiple FPGA Design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, page 175–184, New York, NY, USA, 2012. Association for Computing Machinery.

[35] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs. In *The 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines*. IEEE – Institute of Electrical and Electronics Engineers, May 2015.

[36] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A Configurable Cloud-Scale

DNN Processor for Real-Time AI. In *Proceedings of the 45th International Symposium on Computer Architecture, 2018*. ACM, June 2018.

[37] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. TAPA: A Scalable Task-Parallel Dataflow Programming Framework for Modern FPGAs with Co-Optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.*, 16(4), dec 2023.

[38] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 81–92, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 Gbps Network for HLS. pages 197–203, 08 2021.

[40] Zhenhao He, Daniele Parravicini, Lucian Petrica, Kenneth O'Brien, Gustavo Alonso, and Michaela Blott. ACCL: FPGA-Accelerated Collectives over 100 Gbps TCP-IP. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 33–43, 2021.

[41] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[42] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[43] E.A. Lee and D.G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[44] Michel Lemaire, Daniel Massicotte, and Jean Bélanger. Multi-FPGA Communication Interface for Electric Circuit Co-Simulation. In *2020 IEEE Electric Power and Energy Conference (EPEC)*, pages 1–6, 2020.

[45] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[46] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. CHIP-KNN: A Configurable and High-Performance K-Nearest Neighbors Accelerator on Cloud FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 139–147, 2020.

[47] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–70, 2021.

[48] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware, February 2015.

[49] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking : Bringing Order to the Web. In *The Web Conference*, 1999.

[50] Keshab K Parhi. VLSI digital signal processing systems: design and implementation. In *John Wiley & Sons*, 2007.

[51] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro*, 35(3):10–22, 2015.

[52] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. ZRLMPI: A Unified Programming Model for Reconfigurable Heterogeneous Computing Clusters. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 220–220, 2020.

[53] Manuel Saldana and Paul Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, 2006.

[54] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015.

[55] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanovic, and David Patterson. DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 207–221, New York, NY, USA, 2015. Association for Computing Machinery.

[56] Naif Tarafdar, Nariman Eskandari, Varun Sharma, Charles Lo, and Paul Chow. Galapagos: A Full Stack Approach to FPGA Integration in the Cloud. *IEEE Micro*, 38(6):18–24, 2018.

[57] 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. AIgean: An Open Framework for Machine Learning on Heterogeneous Clusters. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 239–239, 2020.

[58] Xingyu Tian, Zhifan Ye, Alec Lu, Licheng Guo, Yuze Chi, and Zhenman Fang. SASA: A Scalable and Automatic Stencil Acceleration Framework for Optimized Hybrid Spatial and Temporal Parallelism on HBM-Based FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 16(2), apr 2023.

[59] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, feb 2017.

[60] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 93–104, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 845–858, New York, NY, USA, 2020. Association for Computing Machinery.

[62] Yue Zha and Jing Li. Hetero-ViTAL: A Virtualization Stack for Heterogeneous FPGA Clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 470–483, 2021.

[63] Yue Zha and Jing Li. When Application-Specific ISA Meets FPGAs: A Multi-Layer Virtualization Framework for Heterogeneous Cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 123–134, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, page 326–331, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Wentai Zhang, Jiaxi Zhang, Minghua Shen, Guojie Luo, and Nong Xiao. An Efficient Mapping Approach to Large-Scale DNNs on Multi-FPGA Architectures. In *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1241–1244, 2019.