Specializing for Efficiency: Customizing AI Inference Processors on FPGAs

Andrew Boutros[1,2], Eriko Nurvitadhi[2] and Vaughn Betz[1]
[1] ECE Department, University of Toronto and Vector Institute for Artificial Intelligence, Canada
[2] Programmable Solutions Group, Intel Corporation
E-mails: andrew.boutros@mail.utoronto.ca, eriko.nurvitadhi@intel.com, vaughn@eecg.utoronto.ca

***Abstract*—Artificial intelligence (AI) has become an essential component in modern datacenter applications. The high computational complexity of AI algorithms and the stringent latency constraints for datacenter workloads necessitate the use of efficient specialized AI accelerators. However, the rapid changes in state-of-the-art AI algorithms as well as their varying compute and memory demands challenge accelerator deployments in datacenters as a result of the much slower hardware development cycle. To this end, field-programmable gate arrays (FPGAs) offer the necessary adaptability along with the desired custom hardware efficiency. However, FPGA design is non-trivial; it requires deep hardware expertise and suffers from long compile and debug times, making FPGAs difficult to use for software-oriented AI application developers. AI inference *soft processor overlays* address this by allowing application developers to write their AI algorithms in a high-level programming language, which are then compiled into instructions to be executed on an AI-targeted soft processor implemented on the FPGA. While the generality of such overlays can eliminate the long bitstream compile times and make FPGAs more accessible for application developers, some classes of the target workloads do not fully utilize the overlay resources resulting in sub-optimal efficiency. In this paper, we investigate the trade-off between hardware efficiency and designer productivity by quantifying the gains and costs of specializing overlays for different classes of AI workloads. We show that per-workload specialized variants of the neural processing unit (NPU), a state-of-the-art AI inference overlay, can achieve up to 41% better performance and 44% area savings.**

***Index Terms*—FPGA, AI, overlay, soft processor**

I. Introduction

In recent years, AI has become a key ingredient in many end-user applications, such as smart assistants and recommendation systems [1]. The AI algorithms powering these applications are very compute and memory intensive, and therefore are typically executed on remote servers in large-scale datacenters. These workloads usually have tight latency constraints to ensure a pleasant user experience even when multiple algorithms are cascaded, further motivating the need for efficient AI compute. As a result, service providers resort to deploying specialized accelerators to satisfy the increasing demand for faster and more accurate AI processing [2], [3]. However, the rapid evolution of the AI domain is a major challenge for datacenters as the change in AI algorithms and their compute/memory needs is much faster than the typical multi-year datacenter hardware refresh cycle. More frequent hardware changes would not only be very costly, but also challenging for accelerators that use custom-manufactured application-specific integrated circuits (ASICs) due to their slow design and deployment cycle.

The fine-grained reconfigurability of FPGAs offers an appealing solution that combines flexibility and hardware efficiency [4]. An FPGA can be reconfigured on the hardware level to implement different AI algorithms of varying characteristics, enabling a much faster time-to-solution compared to custom ASICs. However, building efficient FPGA systems requires extensive hardware design expertise and suffers from very long compile times (hours to days for large designs) as designs need to be synthesized, placed and routed using complex computer-aided design tools [5]. These two factors create a significant challenge for the typical software-oriented AI application developer when using FPGAs.

Alternatively, FPGA *overlays* follow a similar approach to CPUs or GPUs, in which an instruction set architecture (ISA) layer decouples the hardware and software stacks. This abstracts the hardware complexity away from application developers who describe their algorithms in a high-level programming language. A compiler then translates these programs into sequences of instructions that can run on any processor that supports the same ISA. For the same processor architecture, its implementation as a *hard* ASIC will always be more efficient than as an overlay on the FPGA's reconfigurable logic (i.e. *soft* processor). However, the soft processor implementation can increase efficiency by exploiting the FPGA flexibility to implement a specialized datapath and memory hierarchy. This creates a large design space for FPGA overlays in terms of their degree of specialization. Increasing the overlay generality can enhance productivity by being able to accelerate a wider variety of workloads just by running different (software) instructions on the same soft processor. However, this typically requires over-provisioning the implemented hardware to support all target workloads, resulting in sub-optimal efficiency. On the other hand, a very specialized overlay improves hardware efficiency for a specific workload at the cost of reduced flexibility and longer bitstream compile and FPGA reconfiguration times for other workloads.

In this paper, we study the hardware efficiency vs. designer productivity trade-off of having specialized AI inference overlays for different classes of workloads in contrast to a single overlay that can execute all target workloads. For our study, we use the neural processing unit (NPU), a state-of-the-art FPGA overlay for low-latency AI inference [6]. Our contributions include:

- Implementing a SystemC NPU simulator for fast and accurate performance estimation to enable rapid exploration of the large overlay design space.
- Modifying the NPU architecture to create multiple workload-specialized variants for different classes of workloads.
- Quantifying the area and performance benefits as well as the productivity cost of implementing workload-specialized NPU variants compared to a generic NPU executing all workloads.

II. Background

A. The Neural Processing Unit (NPU)

The NPU is a state-of-the-art very-long-instruction-word (VLIW) processor architecture that resembles the Microsoft Brainwave architecture designed specifically for low-latency AI inference workloads [3], [7]. It keeps all model weights *persistent* in the on-chip memories of one or multiple network-connected FPGAs and

978-1-6654-0839-4/21/$31.00 ©2021 IEEE

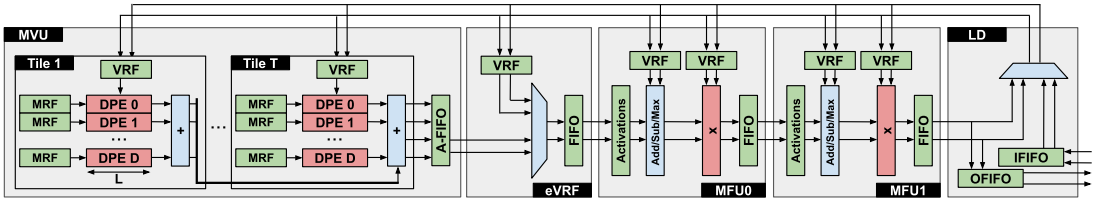

Fig. 1: Overview of the architecture of a single NPU core.

has complex functional units that are well-suited to AI processing. The NPU exploits the abundant data parallelism and deterministic dataflow of AI models by launching thousands of parallel operations with a single instruction, greatly amortizing the energy and area cost of software programmability. Fig. 1 shows the architecture of a single core of our NPU overlay [6]. It consists of five coarse-grained units chained in a pipeline and operates on a batch of 3 independent input streams simultaneously to allow more efficient memory accesses for weights. The core pipeline starts with a matrix-vector multiplication unit (MVU). The MVU consists of T compute tiles, each of which has D dot-product engines (DPEs) of size L multiplication lanes. Vector operands are broadcast from a vector register file (VRF) to all DPEs in a single tile, while persistent model weights come from the matrix register files (MRFs) shared between different NPU cores. The MVU is followed by an external VRF (eVRF) that is used to skip the MVU in case an instruction does not start with a matrix-vector operation. This is followed by two identical multi-function unit (MFU) blocks that implement vector elementwise operations commonly used in AI models, such as activation functions (e.g. sigmoid, tanh, ReLU), addition/subtraction, and multiplication. The final stage is the loader which can write back results to any of the processor architecture states (i.e. VRFs) for further processing, and also can communicate with external components (e.g. other FPGA modules or a network interface) through input/output FIFOs. AI application developers describe their models using a subset of the widely known Tensorflow Keras API [8] which is then compiled into a sequence of NPU VLIW instructions to be executed on the FPGA overlay. This approach abstracts the detailed FPGA hardware away from developers, improving productivity.

B. AI-Optimized Stratix 10 NX FPGA

AI computation is dominated by multiply-accumulate (MAC) operations, so one key to AI performance is maximizing the number of MACs a device can perform. Accordingly, we implement the NPU on Intel's AI-optimized Stratix 10 NX FPGA [9]. Instead of the usual digital signal processing (DSP) blocks that perform a few high-precision MACs (e.g. 2×18-bit MACs on Stratix 10 GX/MX), these FPGAs have tensor blocks that perform a much larger number of low-precision MACs (30× 8-bit) as AI models generally lose little or no accuracy when using lower precisions [10]. Stratix 10 NX not only seeks a large number of MACs per tensor block but also a small enough tensor block that many can be included in a chip; the device we use contains 3960 tensor blocks. One factor in keeping a tensor block small is to use lower precision multiplication, as the area of a multiplier is proportional to the square of its precision. Another concern is the programmable routing (muxing) needed to steer inputs and outputs to and from the tensor block. Stratix 10 NX keeps the cost of this interconnect low by bringing in only ten 8-bit inputs (vs. the 60 required) and using data re-use registers and broadcast lines to feed data to all 30 MACs.

Using the tensor blocks can be challenging for users with limited hardware design expertise. A user can only instantiate this block in a register transfer level (RTL) description like Verilog, and then must design control logic that orchestrates the operation of the tensor block and achieves high compute utilization. Our NPU overlay on Stratix 10 NX hides such low-level design details and allows developers to utilize the tensor blocks purely from software.

TABLE I: Baseline NPU (2C-7T-40D-40L) resource utilization and operating frequency on the Intel Stratix 10 NX device.

Logic (ALMs)	Tensor Blocks	BRAMs	Freq.(MHz)
259,614 (37%)	3,600 (91%)	6,389 (93%)	300

TABLE II: Summary of the specifications of the three AI-optimized devices under study: V100 GPU, T4 GPU, and Stratix 10 NX.

	Nvidia V100	Nvidia T4	Intel S10 NX
On-chip Memory	16 MB	10 MB	16 MB
Process Tech.	TSMC 12nm	TSMC 12nm	Intel 14nm
Die Size (mm^2)	815	545	$<$ 500

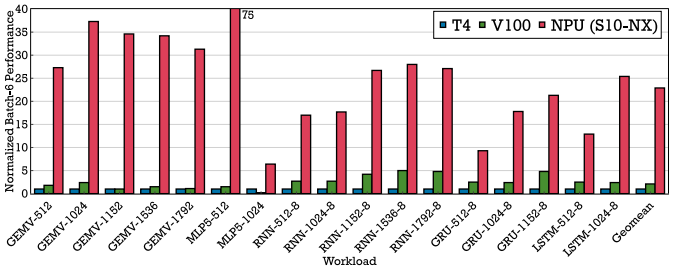

Fig. 2: Performance comparison between NPU on Stratix 10 NX, the V100 GPU, and the T4 GPU on a variety of workloads.

III. Baseline NPU Performance

We implement the baseline NPU architecture with 2 cores, 7 tiles, 40 DPEs, and 40 lanes (2C-7T-40D-40L) on the AI-optimized Stratix 10 NX FPGA and evaluate its performance against same-generation AI-optimized GPUs. Table I presents the resource utilization and operating frequency of the baseline NPU. It utilizes most of the tensor blocks and on-chip block RAMs (BRAMs) while running at 300 MHz, which translates to a peak performance of 40.3 tera operations per second (TOPS). For performance comparison, we use two Nvidia GPUs, the V100 and T4, that use similar process technology, are enhanced with AI-targeted tensor cores, and have similar on-chip memory capacity as shown in Table II. The T4 GPU is more comparable to the Stratix 10 NX FPGA in terms of die size, while the V100 is a much bigger and more powerful GPU. We experiment with a variety of workloads including simple matrix-vector multiplication (GEMV), multi-layer perceptrons (MLPs), and recurrent neural networks (RNNs, GRUs, LSTMs) from the DeepBench suite [11] and Nvidia's persistent RNNs [12]. For every workload, we exhaustively run all possible GPU configurations in terms of numerical format used (32-bit/16-bit floating-point or 8-bit integer), keeping model weights in on-chip memory whenever possible, and tensor core settings. Then, we pick the best measured GPU performance to compare to the NPU on Stratix 10 NX. For a fair comparison, all the GPU results are measured using the `cudaEventRecord()` API which records time stamps on the GPU at the specified points. Therefore, we only account for

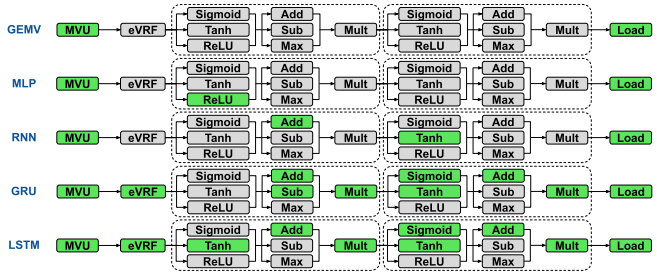

Fig. 3: The usage of the baseline NPU FUs for different classes of workloads. Used FUs are highlighted in green.

the core computation excluding any system-level overheads, which corresponds to the NPU execution cycles on the FPGA.

Fig. 2 shows the performance of the three solutions normalized to that of the T4 GPU for our benchmark suite when processing input batches of size 6 (i.e. running inference on 6 input streams simultaneously). We choose a low batch size since we focus mainly on latency-sensitive applications, in which grouping many inputs before running inference on them might be infeasible. The results show that the NPU performance is significantly higher than both GPUs; it delivers **11**× and **23**× better performance on average compared to the V100 and T4 GPUs, respectively. In addition, all the workloads we experiment with can be written for the NPU in less than 25 lines of Python code, compiled into an NPU instruction stream (a few KBs) in seconds, and then sent to the NPU over PCIe in milliseconds. This highlights that the use of FPGA overlays as a design approach can not only achieve best-in-class AI inference performance, but also make FPGAs more accessible for application developers without long FPGA bitstream compile times or the need for hardware design expertise.

IV. The Gains & Costs of NPU Specialization

The baseline NPU overlay architecture (shown in Fig. 1) has all the function units (FUs) needed to execute the full set of workloads. However, not all of these FUs are utilized when running some workloads. If a certain FU is unused in all NPU VLIW instructions implementing a specific workload, the NPU can still implement this workload even if this FU and its corresponding instruction fields are completely removed from the overlay architecture. Fig. 3 illustrates the used FUs (highlighted in green) for the five main classes of workloads we consider (GEMV, MLP, RNN, GRU, LSTM). In addition, the smaller size variants of these workloads do not need the full capacity of the NPU's register files to store the persistent model weights on-chip. This raises the question of whether higher performance or lower FPGA resource use can be achieved by specializing the NPU for a specific class of workloads by removing unused functional units (grey boxes in Fig. 3). On the other hand, implementing multiple specialized NPU variants requires more design space exploration and more time-consuming hardware compiles. In this section, we investigate the performance and resource gains as well as the productivity cost of implementing specialized NPU variants for these five classes of workloads.

A. SystemC NPU Simulator

To evaluate different NPU variations efficiently, we develop a performance simulator for our NPU overlay architecture in SystemC. Using SystemC allows accurate cycle-level modeling of the NPU architecture details, while still preserving the fast C++ simulation runtime compared to cycle-accurate RTL simulation. This SystemC NPU simulator enables rapid exploration of the NPU design space; we can specify different NPU architecture parameters (e.g. number

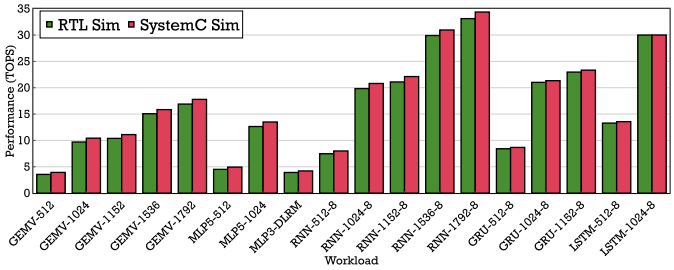

Fig. 4: SystemC NPU simulator performance results compared to cycle-accurate RTL simulation of the NPU Verilog implementation.

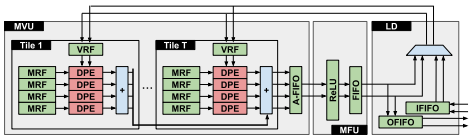

Fig. 5: MLP variant of the NPU architecture.

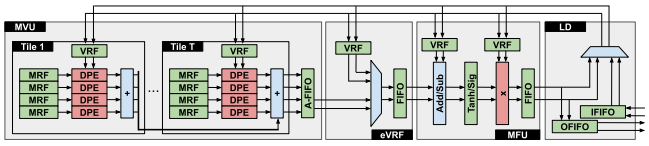

Fig. 6: GRU/LSTM variant of the NPU architecture.

of cores, tiles, DPEs, lanes), different module latencies based on their low-level FPGA implementation, or different compositions of FUs to evaluate their effect on end-to-end workload performance.

Our simulator implementation uses SystemC v2.3.3 and we compare its results and runtime to cycle-accurate RTL simulation of the NPU Verilog using Synopsys VCS v2016.06, with both simulations (SystemC and RTL) running on an Intel Xeon Gold 6146 24-core CPU @ 3.2 GHz with 768 GB of RAMs. All the reported results for the runtime and accuracy comparison are for the 2C-7T-40D-40L baseline NPU architecture. Fig. 4 shows the performance results in TOPS for different workloads reported by our SystemC NPU simulator compared to those from cycle-accurate RTL simulation. Our SystemC simulator accurately estimates NPU performance; its performance predictions are within **5.1%** of the cycle-accurate RTL simulation on average, with a maximum error of 10.8%. This small error results from minor discrepancies between our SystemC and RTL implementations of the NPU architecture that can be tuned to further reduce this gap. The SystemC NPU simulator is **26**× faster than RTL simulation on average, with speedups ranging from 6.5× to 100× for the bigger and smaller workload sizes, respectively. Consequently, our NPU SystemC simulator can be used for rapid yet reliable exploration of the NPU design space.

B. NPU Specialization

The NPU can be specialized to a workload by:

(1) Eliminating unused FUs (grayed out in Fig. 3).
(2) Shuffling the order of FUs in the MFU block to better suit this class of workloads. For example, changing the baseline order {Activation→Add→Mult} to {Add→Activation→Mult}.
(3) Adjusting register file capacities to match the needs of different workload sizes within each class.

Combining (1) and (2) results in four different variants of the NPU core pipeline architecture for GEMV, MLP, RNN, and GRU/LSTM (i.e. both GRUs and LSTMs use the same variant). The MLP and GRU/LSTM variants of the NPU are shown as examples in Fig. 5 and Fig. 6, respectively. Then for each of these variants, we also implement a version with reduced MRF capacity (1 BRAM instead

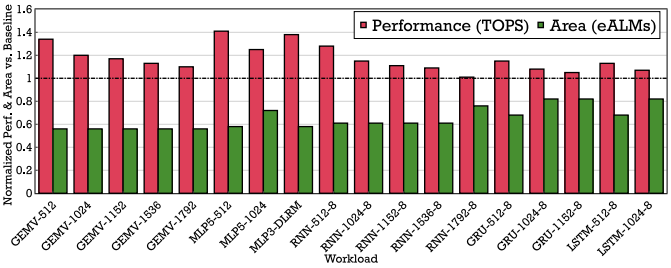

Fig. 7: Performance gains and area savings of NPU specialization.

TABLE III: Average performance gains and area savings of specialization relative to the baseline NPU for different workload classes.

	GEMV	MLP	RNN	LSTM	GRU
Performance	+19%	+35%	+13%	+9%	+10%
Area	-44%	-38%	-36%	-23%	-25%

of 2 for each MRF) which is sufficient for the smaller cases in each of the workload classes.

We use the NPU SystemC simulator to verify the functionality and obtain performance results for each of the NPU variants. Then, we modify the RTL implementation of the NPU to obtain the resource utilization, operating frequency, and bitstream compilation times. All variants are synthesized, placed and routed using Intel Quartus Prime Pro 20.4 on the Stratix 10 NX device. When reporting area results, we use equivalent adaptive logic modules (eALMs) as a representative metric for utilized resources. This metric normalizes different FPGA resource types based on their relative silicon area footprint, where each logic block, BRAM and tensor block is counted as 1, 40 and 33 eALMS, respectively [13].

All four NPU variants close timing at slightly higher than 300 MHz with no significant frequency improvements from specialization. Therefore, when calculating performance, we assume an operating frequency of 300 MHz, matching that of the baseline NPU. This means that any performance gains are a result of the architecture customizations reducing execution cycle counts. Fig. 7 presents the performance and area results of per-workload specialized NPUs normalized to the baseline NPU (dashed line); NPU specialization yields up to **41%** performance improvement and **44%** area savings compared to the baseline architecture. The performance for each workload type improves with a specialized NPU, with higher gains for the smaller workloads whose performance was not limited by the MVU block (e.g. smaller RNNs, GRUs, LSTMs) or workloads with sequential dependencies between layers (e.g. MLPs). Specializing the NPU architecture removes all unused FUs and therefore results in area savings for all workloads. The smaller workloads from each class achieve additional area savings as the model weights can fit in smaller register files. Table III provides a summary of the average performance gains and area savings of NPU specialization for each of the five workload classes compared to the baseline NPU architecture. The achieved area savings can be regarded as a first-order estimates for power savings as well. Another option is to reuse the saved area to fit more MVU compute tiles to further improve performance for MVU-bottlenecked workloads. Our experiments show that we can add two more MVU tiles using the saved resources, which increases the performance of GEMV and RNN workloads by an additional 6% and 9%, respectively.

These performance gains and area savings come at the cost of increased designer effort. Targeting a new workload with the generic NPU only requires writing new software and a fast (seconds) software compile, while specializing an NPU for a new workload requires a slow hardware compile to generate a new FPGA bitstream. The bitstream compilation time of a specialized NPU variant ranged from **230 to 340 minutes**. If the bitstreams of the workloads to be accelerated are precompiled (e.g. in a bitstream library for workloads of interest), however, changing the workload running on the FPGA would only require reconfiguring the FPGA with a different bitstream.

V. Conclusion

The NPU overlay makes FPGAs more accessible for AI application developers by insulating them from both detailed hardware design and long FPGA bitstream compilation times. Once an NPU overlay and its software stack are developed and deployed by a group of experts, application developers can program the FPGA to accelerate different AI workloads purely from software, with seconds-long software recompiles. We showed that by exploiting Intel's AI-optimized Stratix 10 NX device the NPU overlay achieves an order of magnitude better performance than equivalent GPUs, while still being software-programmable. To facilitate design space exploration of alternative NPU architectures, we implemented a SystemC NPU simulator for rapid (26× faster than RTL simulation) and reliable (within 5% of RTL simulation results on average) performance evaluation of NPU architectures on various workloads. Using this simulator, one can rapidly explore the performance gains and area savings of specializing the NPU architecture for different classes of workloads. A per-workload specialized NPU architecture can achieve 9-35% better performance using 23-44% less resources on average. While creating such specialized NPUs requires hours-long FPGA bitstream compilations, if the resulting bitstreams are cached, they can be reconfigured into an FPGA in seconds.

VI. Acknowledgements

The authors thank the NSERC/Intel industrial research chair in programmable silicon and the Intel/VMWare Crossroads 3D-FPGA Research Center for funding support.

References

[1] M. Naumov *et al.*, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *arXiv:1906.00091*, 2019.
[2] N. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Int. Symp. on Computer Architecture (ISCA)*, 2017.
[3] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Int. Symposium on Computer Architecture (ISCA)*, 2018.
[4] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.
[5] K. Murray, *et al.*, "VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling," *Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.
[6] A. Boutros *et al.*, "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs," in *International Conference on Field-Programmable Technology (FPT)*, 2020.
[7] E. Nurvitadhi *et al.*, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *Int. Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.
[8] F. Chollet *et al.*, "Keras," https://keras.io, 2015.
[9] M. Langhammer *et al.*, "Stratix 10 NX Architecture and Applications," in *Int. Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021.
[10] E. Wang *et al.*, "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–39, 2019.
[11] *DeepBench*, (accessed September 24, 2021). [Online]. Available: https://github.com/baidu-research/DeepBench
[12] F. Zhu *et al.*, "Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-chip," *arXiv:1804.10223*, 2018.
[13] A. Boutros *et al.*, "Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.