MPTorch-FPGA: a Custom Mixed-Precision Framework for FPGA-based DNN Training

Sami Ben Ali, Silviu-Ioan Filip, Olivier Sentieys, Guy Lemieux

▶ **To cite this version:**

Sami Ben Ali, Silviu-Ioan Filip, Olivier Sentieys, Guy Lemieux. MPTorch-FPGA: a Custom Mixed-Precision Framework for FPGA-based DNN Training. 28th IEEE/ACM Design, Automation and Test in Europe (DATE), Mar 2025, Lyon, France. pp.1-6. hal-04882989

HAL Id: hal-04882989

https://hal.science/hal-04882989v1

Submitted on 13 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

MPTorch-FPGA: a Custom Mixed-Precision Framework for FPGA-based DNN Training

Sami Ben Ali
Inria, Univ. Rennes

Silviu Filip
Inria, Univ. Rennes

Olivier Sentieys
Inria, Univ. Rennes

***Abstract*—Training Deep Neural Networks (DNNs) is computationally demanding, leading to a growing interest in reduced precision formats to enhance hardware efficiency. Several frameworks explore custom number formats with parameterizable precision through software emulation on CPUs or GPUs. However, they lack comprehensive support for different rounding modes and struggle to accurately evaluate the impact of custom precision for FPGA-based targets. This paper introduces MPTorch-FPGA, an extension of the MPTorch framework for performing custom, multi-precision inference and training computations in CPU, GPU, and FPGA environments in PyTorch. MPTorch-FPGA can generate multiple systolic arrays, each with independent sizes and custom arithmetic implementations that directly provide bit-level accuracy to accelerate GEMM calculations by offloading from the CPU or GPU. An offline matching algorithm selects one of several pre-generated (static) FPGA configurations using a custom performance model to estimate latency. To showcase the versatility of MPTorch-FPGA, we present a series of training benchmarks using diverse DNN models, exploring a range of number format configurations and rounding modes. We report both accuracy and hardware performance metrics, verifying the precision of our performance model by comparing estimated and measured latencies across multiple benchmarks. These results highlight the flexibility and practical value of our framework.**

I. Introduction

Deep Neural Networks (DNNs) have become a cornerstone of modern AI, powering major advancements in fields like image recognition, natural language processing, and autonomous systems. However, the success of DNNs comes with significant computational costs. Training these models is a resource-intensive and demanding process. As DNN models continue to grow in size and complexity, the need for more efficient training methods becomes critical. The use of custom low-precision formats is one promising avenue to improve performance while maintaining accuracy. However, many current frameworks that explore reduced precision formats do so only in a software environment, often without the ability to directly evaluate the effects of custom precision on hardware targets such as FPGAs.

This paper introduces MPTorch-FPGA, illustrated in Figure 1, an extension of the MPTorch framework [*] (Citation omitted for blind review), that integrates custom mixed-precision arithmetic for DNN training in CPU-GPU-FPGA environments. Our framework is designed for researchers and developers who seek to explore and optimize hardware designs with minimal accuracy loss. By combining software emulation and hardware execution within a unified framework, MPTorch-FPGA offers flexibility and precision, allowing for detailed exploration of custom number formats, rounding modes, and arithmetic configurations. The main contributions are:

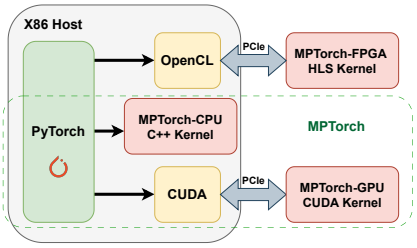

Fig. 1. MPTorch framework overview. It includes CPU, GPU, and the new FPGA component. The MPTorch-FPGA component is the focus of this paper.

- **Unified Emulation and Hardware Framework :** MPTorch-FPGA integrates bit-accurate emulation and hardware exploration, seamlessly combining software and FPGA components. This enables rapid and precise exploration of custom number formats and arithmetic modes. We also provide detailed and fine-grained performance and accuracy benchmarks for various Floating-Point (FP) and Fixed-Point (FXP) configurations across different stages of DNN training.
- **Model-Specific Accelerator Optimization:** Our framework offers a complete flow for selecting an optimized accelerator configuration tailored to each model, optimizing FPGA usage based on the training workload.
- **Efficient Hardware Design:** MPTorch-FPGA uses a single bitstream for each training task, eliminating the need for bitstream switching, and leverages High-Bandwidth Memory (HBM) to efficiently manage multiple cores, addressing FPGA memory constraints in the context of DNN training.

The manuscript is organized as follows. In Sec. II, we discuss previous work, while in Sec. III we discuss the concept and ideas behind MPTorch and MPTorch-FPGA. Sec. IV delves into the implementation aspects of MPTorch-FPGA. Results are presented in Sec. V, followed by a conclusion. The MPTorch-FPGA framework will be released as opensource with all artefacts after acceptance of the paper.

II. Background and Related Work

A. Custom precision for DNN training

Custom arithmetic precision in DNN training seeks to reduce memory consumption while improving computational

efficiency. In this section, we review prominent techniques for custom low-precision training.

General matrix multiplications, the most computationally expensive operations in DNN training, have been a primary focus of precision reduction research [1]–[9]. Many studies explore the use of 8-bit Floating-Point (FP8) formats for GEMM [1]–[5], [9], though accumulations are often still performed with higher 16- or 32-bit precision. To reduce accumulator overhead, other works [5], [9] perform accumulations using lower-precision adders (e.g., FP12). Additionally, [2], [4], [5] use loss scaling and different FP8 formats for the FWD and BWD passes to minimize accuracy loss and memory access overhead.

Beyond FP8, BFloat16 [7], [10] and block floating-point formats [8] have been explored. Some approaches also investigate integer arithmetic [8], [11]–[15]. Stochastic rounding has also been proposed to mitigate rounding errors in low-precision training, in particular due to stagnation [9], [16].

B. Emulation frameworks

Several tools (see Table I) have been proposed in recent years to assess the impact of various hardware and arithmetic choices on DNN training accuracy.

In terms of FPGA-oriented work, Langroudi et al. [17] introduced Cheetah, a co-design framework for DNN inference and training that emulates FP and posit formats using CPU cores and compiles a softcore of Multiply And Accumulate (MAC) operators on FPGA to evaluate hardware characteristics. Tatsumi et al. [5] introduce Archimedes-MPO, a C++-based mixed-precision inference and training framework for FPGAs. Models compiled with Archimedes-MPO use custom templated data types for each low-level operation. These types support up to 32-bit FP or FXP with custom policies including exponent/mantissa/integer/fractional word lengths, optional subnormals, and whether the multiplier output in a MAC is fused or rounded. The FPGA component accelerates GEMM using one single SA synthesized using a single MAC design that supports a fixed mixture of data types and policies.

Table I compares existing frameworks and their features. While frameworks like [5], [17] allow synthesizing custom FPGA operators, their lack of integrated support complicates performance benchmarking across arithmetic configurations. In contrast, MPTorch-FPGA provides a built-in, model-specific FPGA implementation with customizable operators, facilitating easier performance evaluation and supporting various rounding options, including Stochastic Rounding (SR) [18] and Round to Odd (RO) [19].

C. FPGA DNN training accelerators

While simulation frameworks explore precision impacts on accuracy, other research focuses on hardware architectures for custom precision DNN training on FPGAs. Vink et al. [25] introduced the Barista toolchain to simplify DNN accelerator deployment on FPGAs. Integrated with the Caffe framework [26], it allows users to define networks and includes an FPGA-based accelerator with an OpenCL runtime for CNN training. Convolutions use a compile-time sized 2D systolic array, and the authors optimize performance by evaluating various tile sizes to identify the best configuration for specific CNN models. Luo et al. [27] introduced DARK FPGA, a training framework for FPGA accelerators that employs batch-level parallelism and a hardware/software co-design approach for mixed precision training. DARK FPGA enhances computation by determining optimal tile sizes and accelerator parameters.

Using a specific accelerator architecture allows us to tune design parameters for a training workload, while other research focuses on design exploration tools to identify optimal architectures for various workloads. Qi et al. [28] propose a tool for DNN inference and training accelerators that models performance and resource utilization. It employs an exhaustive search method to optimize mappings for each intralayer workload across the architecture space. Similarly, Zhang et al. [29] introduce a design exploration tool for DNN inference accelerators that employs a scalable design paradigm and an automated search process. Their tool divides the exploration into global and local stages to efficiently identify the best accelerator configurations. Additionally, Chen et al. [30] propose an FPGA-based design scheme for CNN accelerators using the roofline model, optimizing computation throughput and memory bandwidth.

Previous works [25], [27], [31] provide user-friendly FPGA acceleration frameworks connected to x86 hosts, facilitating quick testing and integration of their designs. However, they are often difficult to adapt for custom arithmetic configurations and do not investigate the impact of these configurations on overall performance and accuracy. In contrast, our work supports easy evaluation of various arithmetic configurations through a versatile GEMM accelerator.

III. The MPTorch Framework and its FPGA Extension

The goal of MPTorch is to offer a comprehensive resource for researchers investigating DNN acceleration at the arithmetic level, with a strong focus on mixed-precision DNN training. MPTorch includes GPU and CPU components, but this paper focuses on its new FPGA component.

As illustrated in Figure 1, MPTorch is built on top of PyTorch. The CPU and GPU components are designed to offer bit-accurate emulation of arithmetic operations. For the CPU, custom precision operators and quantization functions are implemented in C++, while CUDA is leveraged for GPU implementations. These implementations are exposed to Python via PyBind and seamlessly integrated as PyTorch extensions, ensuring compatibility within the PyTorch ecosystem.

MPTorch-FPGA extends MPTorch with support for FPGA accelerator implementations. FPGA logic is designed using C++ High-Level Synthesis (HLS), controlled through an OpenCL-based Python interface. A detailed discussion of the FPGA accelerator architecture is presented in Section IV.

Figure 2 illustrates the mixed-precision training process in MPTorch-FPGA for a single training iteration. During both forward and backward passes, inputs are quantized to the

TABLE I
COMPARISON OF DIFFERENT DNN TRAINING SIMULATION FRAMEWORKS. MPTORCH-FPGA IS THE ONLY ONE OFFERING MODEL-SPECIFIC ACCELERATOR SUPPORT

Work	AdaPT [20]	ApproxTrain [21]	Cheetah [17]	GoldenEye [22]	QPytorch [23]	FASE [24]	Archimedes-MPO [5]	MPTorch-FPGA Ours
Framework	PyTorch	TensorFlow	TensorFlow	PyTorch	PyTorch	PyTorch,Caffe	TinyDNN	PyTorch
GPU acceleration	✗	✓	✗	✓	✓	✗	✓	✓
Built-in FPGA support	✗	✗	✗	✗	✗	✗	✓	✓
Transformer support	✓	✓	✗	✓	✗	✓	✗	✓
FMA support	✗	✗	✗	✗	✗	✓	✓	✓
Operator emulation	✓	✓	✓	✓	✗	✓	✓	✓
Formats	FXP	FP	Posit,FP	FXP,FP,BFP	FXP,FP,BFP	FP	FXP,FP	FXP,FP
Rounding	-	RZ	RN	RN,RZ	RN,RZ,SR	RN	RN	RN,RZ,SR,RO

desired precision before undergoing matrix multiplication. Our framework emphasizes the impact of custom precisions on GEMM operations[1].

MPTorch-FPGA enables FPGA-based GEMM computation or emulation on GPU/CPU. Results are cast back to full precision. Additionally, the framework supports custom precision simulation for weight updates, where weights are quantized, updated in custom precision, and stored in full precision.

Our framework integrates seamlessly with the PyTorch workflow. Custom-precision operations are easily defined within layer declarations, as illustrated in Figure 3. The parameters for these layers mirror those of standard PyTorch layers, and an extra parameter group to specify the arithmetic configuration for forward and backward passes and quantization formats for executing the layers' GEMM operations.

MPTorch-FPGA provides fine-grained control over arithmetic configurations, allowing independent customization of formats for multiplication and accumulation, precision settings, and rounding modes. It supports both fixed-point, floating-point, and blocked FP arithmetic, offering a variety of rounding modes, including Stochastic Rounding (SR), Round to Nearest Even (RN), Round to Odd (RO), Round to Zero (RZ) and No Rounding (NR), in which the result is calculated exactly.

During emulation (on CPU or GPU), computations are performed using FP32 operators. For both multiplication and addition, pre-quantized inputs are operated on using standard FP32 hardware. The result can be rounded by truncating the least significant bits (RZ mode) or by adding and truncating those bits for RN, RO, and SR. Alternatively, the FP32 result can emulate fused multiply-add (FMA) behavior for low precisions (e.g., 8-bit formats and below).

Emulating custom precision operators introduces significant latency overhead. While this overhead is smaller with GPU implementations, training tasks on CPU can be notably slow. Full precision operators ensure bit-accurate emulation for low-precision formats, though issues like double rounding may arise for higher-precision formats. On the other hand, FPGA implementation provides exact computation using the specified formats and allows for a thorough evaluation of the performance characteristics of custom arithmetic configurations.

[1]Convolution operations are transformed into GEMM computations using the im2col and col2im transformations, performed on the CPU host.

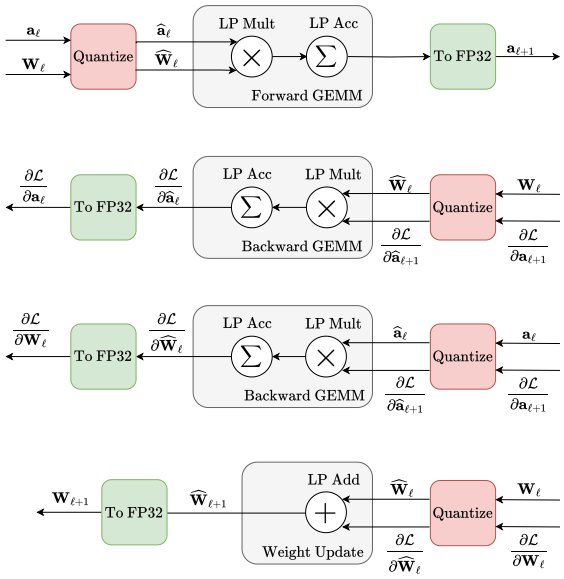

Fig. 2. Low precision computation flow through a linear layer during one iteration of training. The $\mathbf{W}$ variables are the weights, activations are denoted with $\mathbf{a}$, and the loss function with $\mathcal{L}$.

```
import mptorch.quant as qpt

formats = qpt.QAffineFormats(fwd_mac, fwd_rnd, bwd_mac,
                             bwd_rnd, weight_quant,
                             input_quant, grad_quant,
                             bias_quant)

Qlinear_layer = qpt.QLinear(in_features, out_features,
                            formats, device)
```

Fig. 3. MPTorch layer declaration. To enable the FPGA extension, the user should designate an `'fpga'` value to the `device` parameter.

IV. FPGA IMPLEMENTATION

Our FPGA design, which allows for fast and easy benchmarking, has at its core a GEMM accelerator composed of several Systolic Arrays (SAs). Specifically, we use the one-dimensional systolic array introduced by de Fine Licht et al. [32], as shown in Figure 4. This systolic array is composed

of N Processing Elements (PEs), each containing M MAC units. The GEMM computation is performed as inputs are streamed across the PEs. We have significantly modified the original design to accommodate custom arithmetic units and rounding modes. Additionally, our implementation synthesizes multiple SAs within the same chip, as illustrated in Figure 5. This approach addresses the inefficiencies of large tile sizes in conventional SAs, where the tile size is equal to the total number of MAC units $N \times M$. Large tile sizes often result in low utilization for most DNNs, as the input shapes are usually a fraction of the tile size. Moreover, large SAs complicate routing and reduce the design frequency. To mitigate these issues, we deploy multiple smaller SAs, each utilizing separate HBM ports for parallel memory transfer and computation.

A. Performance Model

A training iteration at the layer level consists of a series of GEMM operations. Each GEMM operation input needs to be padded to match the tile size of the selected configuration. By knowing the input shapes and the tiling parameters of the accelerator, we can estimate the latency of each operation. The total training iteration latency is the sum of the latencies of all consecutive GEMM operations.

The input matrices $A \in \mathbb{R}^{n \times k}, B \in \mathbb{R}^{k \times m}$ undergo three stages of padding. The first two are performed on the host CPU before loading the inputs into the FPGA's HBM memory, while the third stage occurs on the FPGA fabric while the data is being loaded from the external memory.

1) **First Stage:** All input dimensions are padded according to the size of memory packs, defined by the number of values that can be stored within one HBM port width (512 bits). For instance, for an 8-bit value, the memory pack size is $512/8 = 64$. The padded dimensions are defined as $n_{\text{mem}}, m_{\text{mem}}$ and k_{mem}.
2) **Second Stage:** the number of cores n_cores should be taken as a divisor of n_{mem}.
3) **Third Stage:** the matrix B (now of dimension m_{mem}) is padded to m_{MAC}, so that m_{MAC} is divisible by the $N \times M$ MAC units within each SA.

The performance of the GEMM operation is calculated using $L_{\text{GEMM}} = \text{Ops/Perf}$, where Ops is the number of multiply and accumulate operations on the padded inputs, i.e.,

$$\text{Ops} = \text{n\_cores} \times m_{\text{MAC}} \times (k_{\text{mem}} \times 2 - 1)$$

and Perf is the peak achievable performance for a given configuration of a systolic array, i.e.,

$$\text{Perf} = N \times M \times F \times 2.$$

F is the frequency of the design, while N, M and C are the size parameters of the accelerator.

The memory transfer between the host CPU and the FPGA off-chip memory is constrained by the PCIe bandwidth and is calculated as

$$L_{\text{data}} = \frac{\text{Size}_{\text{data}}}{B_{\text{PCIe}}}$$

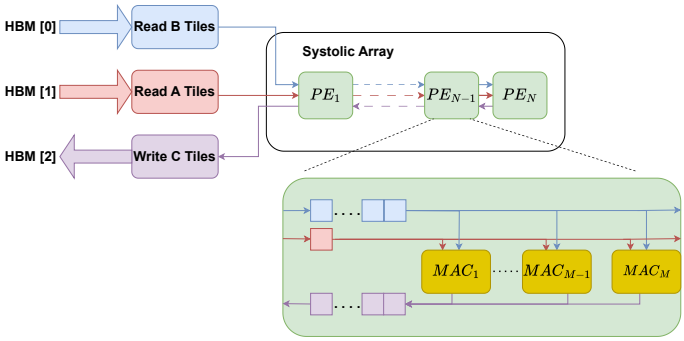

Fig. 4. Systolic array architecture of a GEMM core.

where $\text{Size}_{\text{data}}$ is defined as

$$\text{Size}_{\text{data}} = \text{n\_cores} \times (k_{\text{mem}} + m_{\text{mem}}) + k_{\text{mem}} \times m_{\text{mem}}$$

with L_{data} the data transfer latency, $\text{Size}_{\text{data}}$ the size of the data to be transferred and B_{PCIe} the PCIe bandwidth. In total,

$$L_{\text{total}} = L_{\text{GEMM}} + L_{\text{data}}$$

The latency of memory transfer between FPGA off-chip memory and the FPGA fabric is not considered, as the memory access is done in a pipelined manner with the computation.

B. Accelerator Configuration

To maximize the accelerator's performance, we select the configuration that delivers the best results for a given DNN training workload. We rely on the performance model from Sec. IV-A to accurately estimate the latency of a training iteration for a specific model. We then apply an optimized mapping strategy that ensures the best performance for the DNN model and the accelerator configuration.

Through a unified two-level approach, we minimize latency caused by padding overhead. First, we determine whether to feed the inputs in their original or transposed form. Transposing allows us to switch input dimensions, which can reduce padding overhead. However, this decision is made simultaneously with the second step: optimizing core usage.

For both transposed and non-transposed inputs, we evaluate the latency of GEMM operations across different numbers of cores. Using a brute-force method, we calculate the latency for each combination of input format and core count. The goal is to identify the configuration—whether transposed or not, and with the optimal number of cores—that results in the lowest overall latency.

V. Evaluation

In this section, we illustrate the effectiveness of our framework by exploring how different arithmetic configurations influence model training through bit-accurate emulation. Additionally, we extend this analysis by designing and implementing an FPGA accelerator based on one of the evaluated arithmetic formats. To accomplish this, we apply the methodology from Sec. IV to determine an optimized configuration for the accelerator. Finally, we compare the actual performance of the FPGA accelerator with the predictions made by our

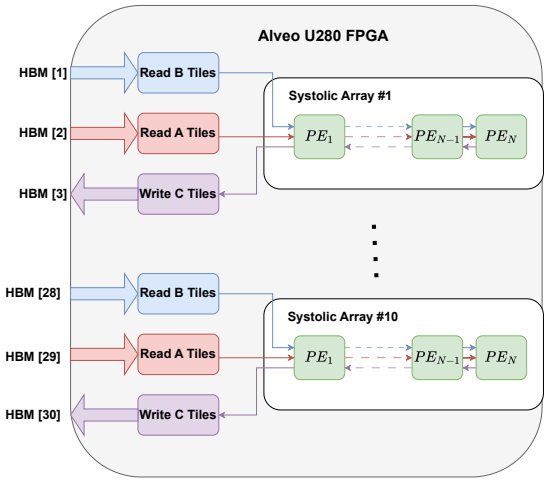

Fig. 5. Accelerator multicore architecture.

TABLE II
TEST ACCURACY COMPARISON ACROSS DIFFERENT MULTIPLIER AND ACCUMULATOR CONFIGURATIONS FOR VARIOUS CNNS.

Multiplier	Accumulator	LeNet5 †	ResNet20 ‡	VGG16 ‡	ResNet50 Δ
E5M2-NR	E6M5-RZ	97.10	10.00	10.00	10.00
	E6M5-RO	98.00	10.00	10.00	10.00
	E6M5-RN	98.61	10.00	10.00	10.00
	E6M5-SR	99.00	90.55	88.99	80.88
	E5M10-RN	99.05	91.24	89.81	82.97
E8M23-RN	E8M23-RN	99.18	91.91	90.67	82.92
FXP4, 4-RN		99.06	10.00	10.00	10.00
FXP4, 4-SR		99.14	10.00	10.00	10.00
FXP4, 4-RZ	FXP8, 8	98.85	10.00	10.00	10.00
FXP4, 4-RO		10.00	10.00	10.00	10.00

Datasets: †MNIST, ‡CIFAR10, ΔImagewoof

model across various DNN training benchmarks, validating the robustness of our approach.

A. Training Setup

We conducted our experiments using various convolutional models, including LeNet5, ResNet20, VGG16, ResNet50, and a Transformer model. The details of the datasets and training configurations for each model are outlined below. In all experiments, we employed adaptive loss scaling [6] with an initial scaling factor of 256 to mitigate accuracy loss during mixed precision training.

1) CNN Experiments: For LeNet5, we performed training on the MNIST dataset for 10 epochs, using a batch size of 64 and a learning rate of 0.1.

In the case of ResNet20 and VGG16, we followed the original training configurations [33], [34]. Both models were trained on the CIFAR10 dataset. For ResNet20, the initial learning rate was set to 0.1, and weight decay was 0.0001, while VGG16 used an initial learning rate of 0.01 and a weight decay of 0.0005. Both models were trained for 200 epochs with a batch size of 128. We used stochastic gradient descent (SGD) with a momentum coefficient of 0.9 throughout.

For ResNet50, we selected the more challenging Imagewoof dataset, a subset of ImageNet that focuses on 10 dog classes out of the 1,000 total classes in the full ImageNet dataset.

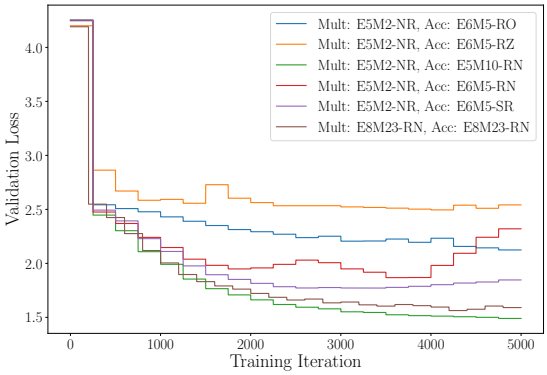

Fig. 6. Nano-GPT validation loss for different arithmetic configurations.

The model was trained with a batch size of 16 and an initial learning rate of 0.01, highlighting the increased computational demand posed by the larger dataset and more complex task.

2) Transformer Experiments: We also conducted training on the Shakespeare dataset, which contains text sequences drawn from Shakespeare's works. We used the Nano-GPT generative model [35] comprising 6 layers, 6 attention heads, a 384 embedding size, and 256 for the block size. Training was carried out for 5,000 iterations using a learning rate of 10^{-4} and the Adam optimizer.

B. Training Results

This section demonstrates the effectiveness of our framework in evaluating the impact of custom precision configurations on DNN model training. Table II presents the emulated test accuracy for various CNN models and arithmetic configurations, while Figure 6 shows the validation loss for the Transformer benchmark. Floating-point formats are denoted as EeMm, where e represents the exponent size and m denotes the mantissa size. For fixed-point formats, the FXP$i.f$ notation is used, with i indicating the size of the signed integer part and f representing the fractional part.

1) Floating Point Formats: We trained models using FMA operators with FP8 multipliers and FP12 or FP16 adders. The FP12 format, previously studied in [5], [9], was further evaluated with different rounding modes. Our results confirm that SR consistently outperforms other rounding modes at the same precision. Although RO and RZ performed well on the LeNet5 benchmark, they failed to converge on other tasks. In [9], the authors demonstrated that increasing the number of random bits can match FP16RN accuracy using FP12SR with 13 random bits. In our experiments, using 10 bits caused slight degradation compared to FP16RN but still provided a notable accuracy advantage over other FP12 configurations.

2) Fixed-Point Formats: We also experimented with FXP8.4 multipliers and FXP16.8 adders. All FXP configurations failed to converge, except for the Lenet5 experiments, where all rounding modes except RO resulted in near FP32 baseline accuracy.

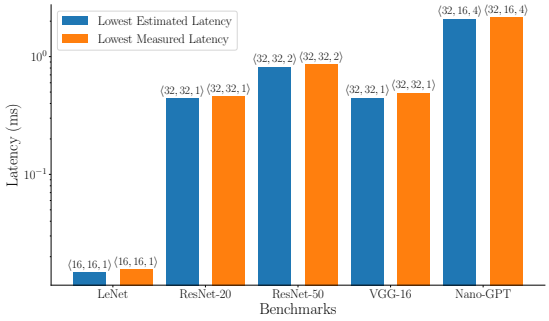

Fig. 7. Lowest Estimated vs Measured Latency and $\langle N, M, C\rangle$ configurations, for Different Training Benchmarks.

C. Accelerator Performance

Our framework facilitates exploration of performance characteristics for different arithmetic formats at the accelerator level. The template-based code allows for easy modification of the accelerator to accommodate custom arithmetic formats or operators. In our experiments, we focus on the FP8 multiplier, FP12SR accumulator FMA configuration, which shows minimal accuracy degradation across all benchmarks. Using the performance model and mapping strategy from Sec. IV, we identify the optimal accelerator configuration for this format. We validate the model's accuracy by comparing its predictions with actual hardware performance.

We use AMD Vitis HLS 2023.1 to synthesize various accelerator configurations, targeting the U55 Alveo FPGA. However, our framework is adaptable to other Alveo and datacenter boards.

In our experiments, we explore the design space by varying the accelerator parameters N, M, and C, subject to specific constraints: N and M must be powers of two, M must be divisible by N, and C is capped at 10 due to the U55's 32 memory ports (with 3 ports per core). We synthesize each configuration at the maximum core count and highest achievable frequency. As detailed in Table III, we present the synthesis results in terms of overall resource utilization on the FPGA, specifically reporting the consumption of Look-Up Tables (LUTs), BRAMs (block RAMS), and Digital Signal Processing blocks (DSPs). The arithmetic operators are implemented using LUTs, while the DSP usage is due to address generation logic within the SAs. As the size of the systolic arrays increases, the number of cores that fit on-chip decreases. The largest systolic array we can accommodate has $N = 64, M = 32$ with $C = 1$.

Using fewer cores can sometimes improve performance due to higher operating frequencies. Table IV presents the estimated training latencies for different models using systolic arrays of size $N = M = 8$, synthesized with varying core counts. The optimal core count is the one resulting in minimum latency.

To identify the optimal accelerator configuration for a training task, we first determine the optimal core count for each SA size. This process is repeated for varying SA sizes to ultimately find the $\langle N, M, C\rangle$ combination that minimizes the estimated training latency.

TABLE III
POSSIBLE ACCELERATOR CONFIGURATIONS (N = #PEs, M = #MACs PER PE, C = #CORES OF SIZE $N \times M$, SHOWING MAXIMAL C AND F, WITH CHIP RESOURCE UTILIZATION).

N	M	C	F (MHz)	LUT (%)	BRAM(%)	DSP(%)
1	1	10	320.9	14.12	13.78	8.56
2	1	10	320.1	14.80	13.80	7.98
2	2	10	320.1	15.10	14.44	8.05
4	2	10	311.0	18.06	15.99	9.76
4	4	10	328.4	21.30	18.20	9.80
8	4	10	197.7	28.20	17.09	11.53
8	8	10	196.2	37.51	21.50	11.53
16	8	10	180.0	61.60	30.3	11.6
16	16	7	160.0	62.73	33.57	7.45
32	16	4	198.4	73.26	33.26	5.72
32	32	2	197.3	62.19	71.48	2.77
64	32	1	150.0	52.57	71.64	1.93

TABLE IV
ESTIMATED TRAINING LATENCY PER ITERATION.

C	F (MHz)	Estimated Training Latency (ms), $N \times M = 8 \times 8$				
		CNN				Transformer
		LeNet5 †	VGG16 ‡	ResNet20 ‡	ResNet50 △	Nano-GPT ◇
1	378.3	0.0273	5.28	1.91	8.15	23.99
2	330.9	0.0211	3.05	1.20	4.70	13.72
3	298.0	0.0189	2.28	0.84	3.62	10.29
4	298.0	0.0173	1.74	0.71	2.64	7.62
5	299.8	**0.0162**	1.43	0.58	2.24	6.15
6	270.6	0.0173	1.32	0.58	2.13	5.67
7	274.7	0.0165	1.19	**0.53**	**1.84**	4.93
8	203.1	0.0218	1.36	0.67	2.14	5.59
9	203.1	0.0215	1.16	0.63	1.90	5.27
10	196.2	0.0218	**1.15**	0.63	1.85	**4.84**

Datasets: †MNIST, ‡CIFAR10, △Imagewoof, ◇Shakespeare

To confirm the accuracy of our performance model, we compare the optimal configurations calculated through the model with those measured on the hardware. Figure 7 shows the latencies of optimal configurations for various training benchmarks. The model successfully identifies all optimal configurations, though measured latencies are slightly higher due to runtime overhead from low-level Xilinx function calls.

VI. CONCLUSION

In this paper, we introduce MPTorch-FPGA, a versatile framework that bridges the gap between software emulation and hardware execution for DNN training. By enabling seamless exploration of arithmetic configurations in FPGA-based systems, it offers researchers a powerful tool to optimize model-specific accelerators and explore precision trade-offs with minimal overhead. We report both accuracy and hardware performance metrics, confirming the precision of our performance model by comparing estimated latency with measured results across various benchmarks. These findings emphasize the flexibility and practical utility of MPTorch-FPGA, demonstrating its effectiveness as a framework for optimizing DNN training across diverse hardware setups. Future extensions of this work could involve finer-grained model matching, such as applying different arithmetic configurations to the forward and backward passes. Additionally, support

for more advanced DNN architectures and larger datasets could further expand the applicability of the framework. The MPTorch-FPGA framework will be released as opensource with all artefacts after acceptance of the paper.

REFERENCES

[1] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu *et al.*, "FP8 formats for deep learning," *arXiv preprint arXiv:2209.05433*, 2022.

[2] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[3] L. Cambier, A. Bhiwandiwalla, T. Gong, M. Nekuii, O. H. Elibol, and H. Tang, "Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks," *arXiv preprint arXiv:2001.05674*, 2020.

[4] N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul, "Mixed precision training with 8-bit floating point," *arXiv preprint arXiv:1905.12334*, 2019.

[5] M. Tatsumi, S.-I. Filip, C. White, O. Sentieys, and G. Lemieux, "Mixing Low-Precision Formats in Multiply-Accumulate Units for DNN Training," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022, pp. 1–9.

[6] P. Micikevicius *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[7] D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[8] S. Q. Zhang, B. McDanel, and H. Kung, "FAST: DNN Training Under Variable Precision Block Floating Point with Stochastic Rounding," in *IEEE Int. Symp. on High-Perf. Comp. Arch. (HPCA)*, 2022, pp. 846–860.

[9] S. Ben Ali, S.-I. Filip, and O. Sentieys, "A Stochastic Rounding-Enabled Low-Precision Floating-Point MAC for DNN Training," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[10] J. Osorio, A. Armejach, E. Petit, G. Henry, and M. Casas, "A BF16 FMA is all you need for DNN training," *IEEE Trans. on Emerging Topics in Computing*, vol. 10, no. 3, pp. 1302–1314, 2022.

[11] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Int. Conf. on Machine Learning*, 2015, pp. 1737–1746.

[12] S.-E. Chang *et al.*, "ESRU: Extremely Low-Bit and Hardware-Efficient Stochastic Rounding Unit Design for Low-Bit DNN Training," in *IEEE/ACM Design, Automation & Test in Europe Conference (DATE)*, 2023, pp. 1–6.

[13] M. Wang, S. Rasoulinezhad, P. H. Leong, and H. K.-H. So, "Niti: Training integer neural networks using integer-only arithmetic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 3249–3261, 2022.

[14] F. Zhu, R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang, and J. Yan, "Towards unified int8 training for convolutional neural network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1969–1979.

[15] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," *arXiv preprint arXiv:1802.04680*, 2018.

[16] P. Blanchard, N. J. Higham, and T. Mary, "A class of fast and accurate summation algorithms," *SIAM journal on scientific computing*, vol. 42, no. 3, pp. A1541–A1557, 2020.

[17] H. F. Langroudi, Z. Carmichael, D. Pastuch, and D. Kudithipudi, "Cheetah: Mixed low-precision hardware & software co-design framework for DNNs on the edge," *arXiv preprint arXiv:1908.02386*, 2019.

[18] M. Croci, M. Fasi, N. J. Higham, T. Mary, and M. Mikaitis, "Stochastic rounding: implementation, error analysis and applications," *Royal Society Open Science*, vol. 9, no. 3, p. 211631, 2022.

[19] S. Boldo and G. Melquiond, "Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 462–471, 2008.

[20] D. Danopoulos, G. Zervakis, K. Siozios, D. Soudris, and J. Henkel, "Adapt: Fast emulation of approximate dnn accelerators in pytorch," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[21] J. Gong, H. Saadat, H. Gamaarachchi, H. Javaid, X. S. Hu, and S. Parameswaran, "ApproxTrain: Fast Simulation of Approximate Multipliers for DNN Training and Inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[22] A. Mahmoud, T. Tambe, T. Aloui, D. Brooks, and G.-Y. Wei, "Goldeneye: A platform for evaluating emerging numerical data formats in dnn accelerators," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 206–214.

[23] T. Zhang, Z. Lin, G. Yang, and C. De Sa, "Qpytorch: A low-precision arithmetic simulation framework," in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE, 2019, pp. 10–13.

[24] J. Osorio, A. Armejach, E. Petit, G. Henry, and M. Casas, "FASE: A fast, accurate and seamless emulator for custom numerical formats," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2022, pp. 480–497.

[25] D. A. Vink, A. Rajagopal, S. I. Venieris, and C.-S. Bouganis, "Caffe barista: Brewing caffe with fpgas in the training loop," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 317–322.

[26] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[27] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, and C. Guo, "Towards Efficient Deep Neural Network Training by FPGA-Based Batch-Level Parallelism," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 45–52.

[28] Y. Qi, S. Zhang, and T. M. Taha, "TRIM: A Design Space Exploration Model for Deep Neural Networks Inference and Training Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 5, pp. 1648–1661, 2022.

[29] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[30] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[31] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-CNN: An FPGA-based framework for training convolutional neural networks," in *2016 IEEE 27Th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2016, pp. 107–114.

[32] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 244–254.

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

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

[35] nanoGPT Contributors, "nanoGPT," 2023, accessed: 2024-09-06. [Online]. Available: https://github.com/karpathy/nanoGPT