White Paper

Author's version of paper published in 2020 IEEE International Conference on Field Programmable Technology (FPT)

Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs

Andrew Boutros[1,2], Eriko Nurvitadhi[1], Rui Ma[1], Sergey Gribok[1], Zhipeng Zhao[3], James C. Hoe[3], Vaughn Betz[2], Martin Langhammer[1]

[1]Programmable Solutions Group, Intel Corporation

[2]University of Toronto and Vector Institute [3]Carnegie Mellon University

{andrew.boutros, eriko.nurvitadhi}@intel.com

***Abstract*—The growing importance and compute demands of artificial intelligence (AI) have led to the emergence of domain-optimized hardware platforms. For example, Nvidia GPUs introduced specialized tensor cores for matrix operations to speed up deep learning (DL) computation, resulting in very high peak throughput up to 130 int8 TOPS in the T4 GPU. Recently, Intel introduced its first AI-optimized 14nm FPGA, the Stratix 10 NX, with in-fabric AI tensor blocks that offer estimated peak performance up to 143 int8 TOPS, comparable to 12nm GPUs. However, what matters in practice is not the peak performance but the actual achievable performance on target workloads. This depends mainly on the utilization of the tensor units, and the system-level overheads to send data to/from the accelerator.**

This paper presents the first performance evaluation of Intel's AI-optimized FPGA, the Stratix 10 NX, in comparison to the latest accessible AI-optimized GPUs, the Nvidia T4 and V100, on a large suite of real-time DL inference workloads. We enhance a re-implementation of the Brainwave NPU overlay architecture to utilize the FPGA's AI tensor blocks, and develop toolchain support that allows users to program tensor blocks purely through software, without FPGA EDA tools in the loop. We first compare the Stratix 10 NX NPU against Stratix 10 GX/MX versions with no tensor blocks, and then present detailed core compute and system-level performance comparisons to the T4 and V100 GPUs. We show that our enhanced NPU on Stratix 10 NX achieves better tensor block utilization than GPUs, resulting in 24× and 12× average compute speedups over the T4 and V100 GPUs at batch-6. Even with relaxed latency constraints that allow a batch size of 32, we still achieve average speedups of 5× and 2× against T4 and V100 GPUs, respectively. On a system-level, the FPGA's fine-grained flexibility with its integrated 100 Gbps Ethernet allows for remote access at 10× and 2× less system overhead latency than local access to a V100 GPU via 128 Gbps PCIe for short and long sequence RNNs, respectively.

***Index Terms*—FPGA, GPU, Deep Learning, Neural Networks**

I. Introduction

The rapid advances in deep learning (DL) now offer unprecedented quality of results in a growing number of application domains, such as robotics [1], natural language processing [2], and complex strategy games [3], [4]. These advances have also opened the door for a myriad of end-user commercial applications. Major tech companies, such as Microsoft, Google and Facebook, now offer a variety of DL-based intelligent services [5]–[7]. To deliver better quality of results, improved DL algorithms continue to demand more compute, which poses a great challenge especially in combination with the strict latency constraints of these applications. This has led to myriad hardware innovations and diverse solutions for AI-optimized computing. For example, Nvidia enhanced its GPU microarchitecture to tightly integrate tensor cores, which are specialized units for matrix operations targeting DL workloads. Such units offer high tensor arithmetic throughput, resulting in a substantial increase in the GPU's peak tera operations per second (TOPS). More recently, many different AI ASICs have been announced, such as Groq's Tensor Streaming Processors [8] and Graphcore's Intelligence Processing Unit [9], that promise even higher peak performance of up to 820 `int8` TOPS [10].

For FPGAs, several proposals to improve the peak device throughput have coarsely integrated an FPGA fabric with a separate AI-optimized compute complex, such as in the Xilinx Versal architecture [11] or AI-targeted chiplets in Intel's system-in-package ecosystem [12], [13]. More recently, Intel introduced its first AI-optimized FPGA, the Stratix 10 NX, which integrates new AI tensor blocks and delivers up to 143 `int8` and block `fp16` TOPS [14]; a comparable peak performance to similar-generation GPUs. Unlike prior approaches, the NX AI tensor blocks are tightly integrated in the FPGA fabric, similar to standard FPGA DSP blocks. This tight integration allows for a richer and more flexible connectivity to the programmable fabric, and can still be programmed using the standard FPGA development flow.

With the race towards incorporation of tensor compute into AI-optimized hardware, the peak TOPS number is used as a key metric when comparing potential acceleration solutions. However, this can be misleading since the peak performance is only attainable when the tensor units are 100% utilized, which is usually not the case in real applications. The utilization of the tensor compute units is typically affected by two main factors: the mapping of a given workload to the available compute units, and the end-to-end system-level overheads of bringing the data in/out of the chip. This work studies the *actual achievable* performance of AI-optimized FPGAs and GPUs through detailed evaluation of both core compute and system-level performance on key AI workloads.

To enable the FPGA evaluation, we enhance the design of Microsoft's Brainwave NPU [5], a state-of-the-art commercial AI soft processor, to efficiently use the tensor blocks in the recently announced Stratix 10 NX. We then compare our enhanced NPU implemented on Stratix 10 NX against the baseline NPU from [13] on Stratix 10 GX/MX devices with no tensor blocks. Finally, we compare the enhanced NPU on NX against the latest accessible AI-optimized Nvidia GPUs with tensor cores, the T4 and V100[1], and study their real performance on a large suite of real-time DL inference workloads, such as: MLPs, RNNs, GRUs, and LSTMs. The contributions of this paper include:

- Architecture, instruction set and compiler enhancements to the NPU overlay to utilize the Stratix 10 NX tensor blocks.
- Performance comparison of our enhanced NPU against the prior baseline NPU on standard FPGAs without tensor blocks.
- Evaluation of actual achievable performance of AI-optimized FPGAs and GPUs on diverse real-time DL workloads.
- Characterization of system-level overheads in both Ethernet-connected FPGAs and PCIe-connected GPUs.

[1]Nvidia has recently announced a 7nm A100 device with higher throughput, but it is not yet generally available so we cannot benchmark it, and it is also not a comparable device in terms of process technology.

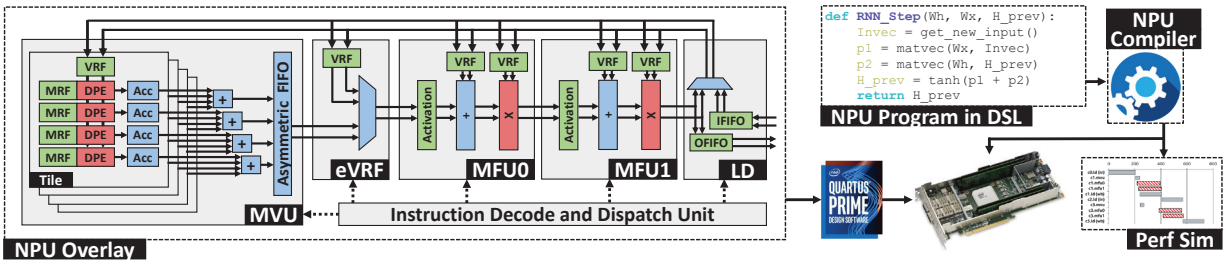

Fig. 1: Overview of the baseline NPU overlay architecture and the front-end tool chain for programming the NPU soft processor.

II. Background

A. DL Workloads: MLPs and RNNs

We study real-time DL inference using multi-layer perceptrons (MLPs) and recurrent neural networks (RNNs), which represented 90% of Google's DL datacenter workloads in 2017 [6]. With the emergence of newer and more sophisticated DL algorithms, these models still constitute a significant portion of DL workloads since many of the newer models use MLPs and RNNs as subroutines [15], [16]. MLPs [17] are arguably the simplest form of neural networks, as they consist of stacked fully-connected layers with non-linear activation functions in between. They are pervasively used in many applications and represent an essential component of Facebook's most recent recommendation models [18].

On the other hand, RNNs are models that process sequence inputs such as speech samples or sentences. They consist mainly of a number of matrix-vector multiplications followed by vector element-wise operations that form *gates*. In this work, we use 3 different variations of RNNs: vanilla RNNs, gated recurrent units (GRUs), and long short-term memories (LSTMs) with 2, 6, and 8 vector-matrix multiplications per time step, respectively [19]–[21]. The following formulas describe the computation of LSTMs.

$$\begin{aligned} i_t&=\sigma(x_tW_i+h_{t-1}U_i+b_i) & f_t&=\sigma(x_tW_f+h_{t-1}U_f+b_f)\\ o_t&=\sigma(x_tW_o+h_{t-1}U_o+b_o) & g_t&=\tanh(x_tW_g+h_{t-1}U_g+b_g)\\ c_t&=f_t\circ c_{t-1}+i_t\circ g_t & h_t&=o_t\circ\tanh(c_t) \end{aligned}$$

where $x_t, i_t, f_t, o_t, g_t, c_t, h_t$ are the input, input gate, forget gate, output gate, cell input, cell state and hidden state vectors at time step t, respectively. The $\circ$ and $+$ operators denote vector element-wise multiplication and addition, while σ and $\tanh$ are the sigmoid and hyperbolic tangent activations. W and U are the input and hidden matrices, and the b terms are bias vectors. RNNs are typically used in speech analysis and natural language processing, and are part of the most recent MLPerf v0.7 inference benchmark suite [22]. In this paper, we specify RNNs with their sizes and number of steps. An LSTM-1024-16 workload describes an LSTM with eight 1024×1024 matrices and 16 time steps. The RNNs we use are from DeepBench [23] and Nvidia's persistent RNNs [24].

B. Baseline NPU Architecture and Toolchain

The Brainwave NPU is an FPGA-based AI overlay that was designed by Microsoft targeting low-latency batch-1 DL inference [5]. It keeps all the model weights persistent on one or multiple network-connected FPGAs to eliminate any external off-chip memory accesses that can increase the processing latency. Our baseline NPU is a re-implementation of Microsoft's Brainwave NPU based on the published description in [5] and [25]. However, we implement standard `int8` precision instead of Microsoft's custom floating point formats (e.g. `fp11` and `fp8`) for a more direct comparison to other hardware platforms. The performance of our re-implementation closely matches that of the Microsoft Brainwave when using the same NPU configuration. Fig. 1 gives an overview of our NPU architecture that has 5 main pipeline stages. The matrix-vector multiplication unit (MVU) consists of T tiles followed by an inter-tile adder reduction tree. Each tile contains a vector register file (VRF) to store the input vectors and D dot product engines (DPEs), each of which has L multiplication lanes. A DPE is tightly coupled with a matrix register file (MRF), that stores the persistent model weights, and a local accumulator. The output bandwidth of the MVU is reduced from D to L vector elements using an asymmetric FIFO before exiting the MVU block. The external VRF (eVRF) block enables skipping the MVU for instructions that do not have a matrix-vector operation. The eVRF is followed by two multi-function units (MFUs) for vector element-wise activation, addition and multiplication. Finally, the loader (LD) block communicates with the outside world via the input/output FIFOs, and can write back the pipeline results to any of the architecture's VRFs. This architecture uses VLIW instructions of 5 macro-operations (mOPs), one for each stage of the pipeline. An mOP gets decoded into a sequence of micro-operations (μOPs) and issued to the different pipeline stages. The NPU illustrated in Fig. 1 as an example has 4 tiles, 4 DPEs and 2 lanes. We refer to such configuration as 4T-4D-2L and use this notation to describe NPU configurations in the rest of this paper.

We also implement a complete NPU software toolchain in which an application developer can write NPU programs in a domain-specific language (DSL) as shown in Fig. 1. The NPU program is compiled into NPU VLIW instructions, that can be simulated in our C++ cycle-accurate performance simulator, or used to program a deployed NPU instance. This approach allows application developers to rapidly experiment with different NPU programs without needing any FPGA design expertise or suffering from the long compile time of FPGA CAD tools. With minor modifications based on [26], our NPU architecture and toolchain can support convolutional neural networks which represent another important class of DL workloads for computer vision applications. However, we leave that for future work and focus mainly on the workloads we discussed in this section. We refer the reader to [5] and [13] for complete details about the NPU overlay and toolchain.

III. Stratix 10 NX: An AI-Optimized FPGA

A. Trends in FPGA Architecture for DL

The growing computational demands and ubiquity of DL motivates adding more compute units, particularly low-precision integer multiply-accumulates (MACs) for inference. The authors of [27]–[29] propose several logic block modifications that can increase the density of soft low-precision multipliers. Other work focuses on enhancing the fracturability of multipliers in DSP blocks to increase low-precision MAC density while keeping the same block interface to avoid adding costly routing ports. In [30], an Intel-like DSP block is modified to support more `int9` and `int4` operations per block with negligible die size increase. In the same vein, the next-generation Intel Agilex family [31] adds a DSP mode with four `int9` multiplications per block, while the PIR-DSP block [32] enhances a Xilinx-like DSP block by incorporating low precisions with interconnect and data reuse optimizations. Some prior work [33], [34] also explore the idea of integrating specialized DL hard blocks in the FPGA fabric, but none of them showed any experimental results on real DL acceleration architectures. The recently announced Stratix 10 NX FPGA, with new AI tensor blocks, drops the legacy DSP block

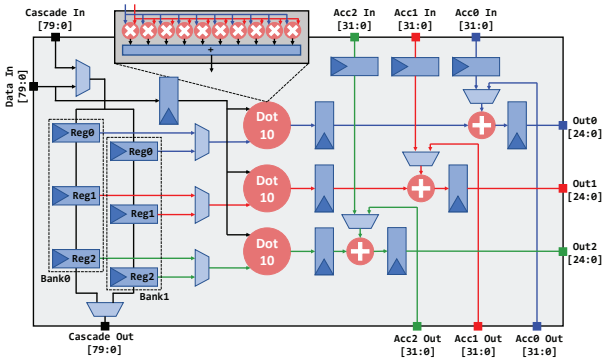

Fig. 2: Simplified block diagram of the `int8` tensor mode of the Stratix 10 NX AI tensor block.

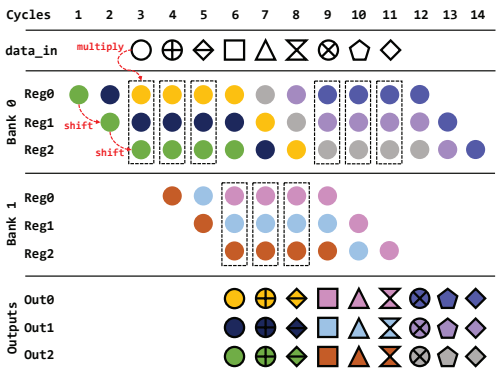

Fig. 3: Tensor mode operation of the Stratix 10 NX AI tensor block. Shapes represent different input data vectors broadcast to the three dot product units, and colors represent vectors shifted into the ping-pong register banks via input cascade from the block above. Dashed boxes show the register bank fed to the dot products at each cycle to produce an output of the corresponding shape and color.

modes used to implement efficient filtering structures, and replaces them with modes and precisions tailored for AI workloads while maintaining about the same die size [35]. To alleviate the routing ports limitation, the new tensor block introduces data reuse register banks that enable fitting a significantly higher number of `int8` multipliers than any prior work, while keeping the same number of input/output ports as the standard Stratix 10 DSP block. Another line of work proposes integrating the FPGA fabric side-by-side with a separate AI compute complex, such as the AI engines in Xilinx's Versal architecture [11] and specialized AI chiplets using Intel's system-in-package ecosystem [12], [13]. In contrast, Stratix 10 NX integrates in-fabric tensor blocks, which enables more flexible connectivity to the soft fabric and uses the existing FPGA development flow.

B. The Stratix 10 NX AI Tensor Block

The Stratix 10 NX tensor block is a configurable function block designed specifically for AI workloads. It replaces the conventional Stratix 10 variable precision DSP block [36], and supports scalar, vector and tensor modes of operation with various numerical formats such as integer (`int8`/`int4`), single-precision floating point (`fp32`), block floating point, and brain floating point (`bfloat24`/`bfloat16`). To enable a very high multiplier count per tensor block without additional routing ports, Stratix 10 NX uses a mix of data broadcast and serially loaded data re-use registers. To best utilize this block, one must carefully consider how to map an application's computation to it so enough operands are available to keep all the multipliers busy each cycle. Next, we focus specifically on the `int8` tensor mode which we use in this paper. We refer to [35] for more details on the Stratix 10 NX device.

We recreate Fig. 2 as a simplified version of the Stratix 10 NX AI tensor block diagram from the Intel technology brief [14] to highlight the `int8` tensor mode and how we use it. The tensor block contains three dot product units, each of which has ten 8×8 multipliers, and three optional accumulators. Two banks of ping-pong data reuse registers (preload buffers) are used to store operands and can be populated through either the block's data input port (via its ten 8-bit connections to the programmable routing) or through a dedicated chain from the tensor block above. Typically, one tensor block at the beginning of each chain is used as an input bypass to load inputs to the ping-pong register chain of the tensor blocks below. For the rest of the tensor blocks in the chain, each dot product unit receives one set of (thirty) operands from a bank of ping-pong registers and another set of (ten) operands is broadcast to all three dot product units directly from the data input port. While the first bank of registers is feeding operands to the dot product units, the second bank can be loaded (over 3 cycles) from the tensor block above. Dedicated hard chains between accumulators can also be used to cascade multiple tensor blocks in a column to form longer dot product units efficiently. Fig. 3 illustrates the operation of the AI tensor block in this mode, where colored circles are input vectors shifted into the ping-pong register banks and uncolored shapes are input vectors to the data port of the tensor block broadcast to the three dot product units. The bank of registers feeding the dot product units each cycle is marked with a dashed box. After 3 clock cycles, the outputs drawn using the shape and colors of their corresponding operands are produced.

IV. Enhanced NPU for Stratix 10 NX

A. NPU Architecture & Toolchain Enhancements

1) Matrix-Vector Multiplication Mapping: Fig. 4a illustrates the mapping of the matrix-vector operation to an MVU with two tiles and DPEs in the baseline NPU. The matrix is split horizontally into T *column blocks* such that each tile is responsible for one column block. Each DPE in a tile performs dot product between a block of an input vector and a matrix row block over multiple cycles. Then, outputs of corresponding DPEs from different tiles are reduced to produce the final result before progressing to the next *row block* of the matrix. To leverage the new tensor blocks in Stratix 10 NX, we must reorganize the NPU's MVU. We load blocks of 3 different input vectors (to feed the 3 dot product units) to one of the register banks (B0) and reuse them across enough matrix rows to hide the latency of loading the next 3 blocks of the input vectors to the other register bank (B1), as shown in Fig. 4b. Recall that B0 and B1 correspond to Bank0 and Bank1 in Fig. 2. This means that the architecture is working on a batch of 3 inputs. However, this leads to interleaving the accumulation of multiple partial results (shown as dash-outlined orange, yellow and violet results in $t = 1, 2, 3$), and requires redesigning the accumulators as discussed later in this section.

An alternative mapping would be to shift matrix blocks into the register banks and broadcast vector blocks instead. However, as we cascade more tensor blocks, the latency of loading to the chained register banks becomes longer, and therefore requires a larger batch size to hide. For example, with 40-lane DPEs, we would need to cascade 4 tensor blocks and have at least a batch size of 12 input vectors to hide the loading latency to a chain of 12 registers (3 registers per block). Thus, we use the first mapping (in Fig. 4b) to keep a low batch size and use matrix rows, which are typically a large number, as broadcast inputs to hide the loading latency.

2) Accumulator Design: As a result of interleaving the accumulation of multiple partial results as illustrated in Fig. 4b, the MVU accumulators can no longer be a simple register and adder. Instead, we implement a BRAM-based accumulator that functions as a small scratchpad for storing multiple interleaved partial results feeding a single adder, as shown in Fig. 4c. The accumulation is

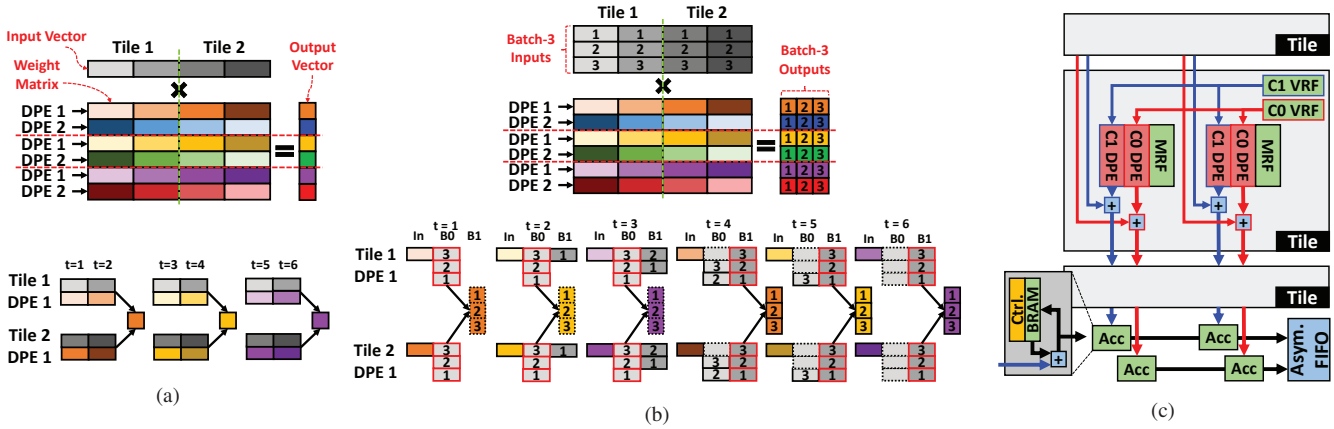

Fig. 4: Matrix-vector multiplication mapping in the (a) baseline NPU and the (b) enhanced NPU on Stratix 10 NX. (c) MVU architecture in our enhanced NPU with BRAM-based accumulators, daisy-chain interconnect, and 2 cores sharing the same MRFs.

done in `int32` precision, and therefore only a few BRAMs are needed to store the partial results. We also move the accumulators after the inter-tile reduction (see Fig. 1), such that we reduce, then accumulate. This amortizes the cost of the accumulators across all tiles instead of having a different set of accumulators for each tile as in the baseline NPU [13].

3) Daisy Chain Tiles: The output bandwidth of the MVU tiles is tripled compared to the baseline NPU as a side-effect of operating on a batch of 3 input vectors. As a result, routing wide buses from each tile to a central adder tree to perform the inter-tile reduction can cause substantial routing congestion. For example, an MVU tile with 40 DPEs, each producing three 32-bit outputs, results in a $3,840$-bit wide bus going from each tile to the central adder reduction tree. To mitigate the resulting routing congestion, we redesign the MVU to have a daisy-chain architecture in which each tile gets the results of the previous tile, performs a local binary reduction and passes the results to the next tile, as shown in Fig. 4c. This architecture uses shorter and more localized routing between each two consecutive tiles and is found to be more efficient and routing-friendly [37], at the cost of a few cycles higher latency.

4) Multi-Core NPU: To choose the configuration of our enhanced NPU, we use our cycle-accurate performance simulator to rapidly explore the design space. We find that the mismatch between the MVU output bandwidth ($3 \times D$ elements) and the rest of the pipeline blocks (L elements) can become the performance bottleneck if D is significantly larger than L, where D and L are the numbers of DPEs and lanes, respectively. Therefore, we decide to keep $D = L$ to minimize the bandwidth mismatch without spending a significant amount of logic and routing resources to increase the bandwidth of all the pipeline blocks. We also find that $L = 40$ offers a good tradeoff between the number of tensor blocks used as input bypass (1 for every $\lceil L/10 \rceil$ chained tensor blocks) and the number of cycles needed to load the chain of ping-pong registers ($3 \times \lceil L/10 \rceil$ cycles). Therefore, to scale up our NPU architecture, this leaves only the number of tiles T as a free parameter. However, introducing too much parallelism only in one dimension can result in underutilized hardware, especially for smaller workloads. As a result, we introduce the concept of *cores* to the NPU architecture. An NPU core is a complete NPU pipeline with all 5 blocks in Fig. 1, except that all cores share the same MRFs holding the persistent model weights and execute the same instruction stream in a single-instruction multiple-thread (SIMT) fashion. Based on our design space search, we implement an NPU architecture with 2 cores, 7 tiles, 40 DPEs, and 40 lanes (2C-7T-40D-40L), which offers a good tradeoff between hardware utilization, processing latency, and FPGA resource utilization. In this configuration, each core processes a batch size of 3 in parallel, resulting in an effective batch size of 6. The new MVU architecture of the enhanced NPU is shown in Fig. 4c.

TABLE I: Implementation results for the largest NPU overlay instance on Stratix 10 NX, MX and GX devices (TBs: Tensor Blocks).

	S10 NX	S10 MX 2100	S10 GX 2800
NPU Config.	2C-7T-40D-40L	1C-4T-80D-40L	1C-4T-120D-40L
# Multipliers	67,200	12,800	19,200
ALMs	256,125 (37%)	399,506 (57%)	567,982 (61%)
BRAMs	6,400 (93%)	6,428 (94%)	9,018 (77%)
TBs/DSPs	3,600 (91%)	3,360 (85%)	4,880 (85%)
Freq. (MHz)	300	290	275
Peak int8 TOPS*	40.3	7.4	10.6
TOPS/Mil.LEs	19.45	3.58	3.84

* NPU peak TOPS is calculated using only the TBs/DSPs in the MVU which are 2240, 3200 and 4800 in the NX, MX and GX versions, respectively. Other TBs/DSPs are used in the MFU or as input bypass in NX.

5) ISA & Toolchain: The enhanced NPU also introduces new instructions for batch-3 operations. The mOP definition of all the pipeline blocks now have 3 fields for addresses of 3 operand vectors instead of 1 in the baseline NPU. In addition, the MVU μOP definition is changed to add the low-level control signals of the tensor blocks such as register enables for the ping-pong register banks, and mux select signals to choose the set of inputs to feed the dot product units. The C++ performance simulator is also modified to reflect the new architectural changes for rapid, yet accurate, performance estimation. Finally, we add support to the DSL and NPU compiler for batch operations. This approach allows the application developer to program the FPGA's tensor blocks purely in software without the need to worry about operation mapping and low-level control sequencing for the tensor blocks.

B. Implementation Results

Table I presents the implementation results of our enhanced NPU on the Stratix 10 NX device in comparison to our baseline NPU from [13] on both the MX 2100 and GX 2800 devices with no tensor blocks. We use the fastest fabric speed grade for all three devices and report results from Intel Quartus 20.1 with device support for the Stratix 10 NX provided by Intel. Our enhanced NPU on NX runs at a slightly higher frequency, and achieves 5.25× and 3.5× higher multiplier count compared to the NPU on a (similarly sized) MX and a (larger sized) GX devices, respectively. Out of the 3600 tensor blocks utilized by the NPU on NX, only 2240 (57% of the device's tensor blocks) are used in tensor mode by the MVU; we only count these blocks to compute 40.3 peak TOPS, as this best lines up with GPU and baseline NPU practices. The remaining tensor blocks are used to implement the MFU's `int32` element-wise multiplications (800 blocks) or used as input bypass to feed the data reuse registers of a chain of tensor blocks

TABLE II: Batch-6 performance results of our enhanced NPU on Stratix 10 NX for different workloads (MLP-5: five-layer MLP).

Workload	Size (h)	Steps (t)	Latency (ms)	Eff. int8 TOPS	HW Util.
MLP-5	512	–	0.004	4.3	10.7%
	1024	–	0.005	12.3	30.4%
RNN	512	256	0.23	7.0	17.4%
	1024	256	0.33	19.1	47.2%
	1152	256	0.39	20.4	50.7%
	1536	256	0.49	29.1	72.2%
	1792	256	0.61	32.4	80.3%
GRU	512	256	0.59	8.1	20.1%
	1024	256	0.95	20.4	50.7%
	1152	256	1.1	22.6	55.9%
LSTM	512	256	0.51	12.7	31.6%
	1024	256	0.89	29.0	71.9%

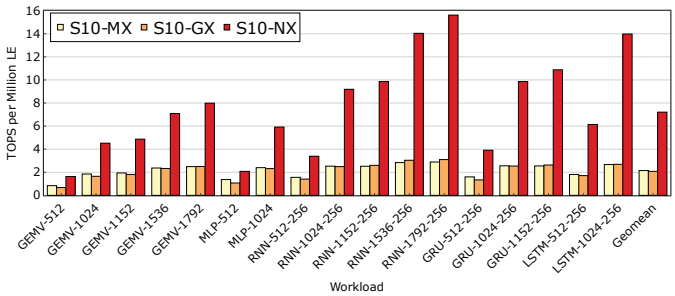

Fig. 5: Effective `int8` TOPS per million LE of the NPU on the Stratix 10 MX, GX and NX devices for GEMV and DL workloads.

through the input cascades (560 blocks). This is an artifact of our NPU that is tailored for low-batch persistent AI use cases. Other applications can use longer chains with only one input bypass tensor block per clock sector. In addition, if the application is not bound by BRAM utilization, vector element-wise multiplications can still be implemented efficiently in the soft logic [38], while the tensor blocks are freed up for more dense compute operations. In the MX and GX NPU versions, we implement dense packing of four `int8` multipliers with shared inputs per DSP block, using the two 18-bit hard multipliers with soft logic for result contamination recovery and dot product reduction [39]. The tensor blocks with native `int8` support eliminate the use of soft logic in implementing the DPEs. This, along with other microarchitectural optimizations, reduces soft logic utilization by 36% and 55% compared to the MX and GX implementations, respectively.

Table II shows the batch-6 latency, effective performance and hardware utilization results of our enhanced NPU on the Stratix 10 NX device on real-time DL workloads. We experimentally measure the NPU performance on a Stratix 10 NX development kit using hardware timestamps to count the number of cycles starting from consuming the inputs from the input FIFO, executing all NPU instructions and writing the outputs to the output FIFO. We also verified that these measurements exactly match the expected number of cycles from RTL simulation. The results show that the effective performance of our NPU increases with increasing problem size, achieving up to **32.4 TOPS** with **80.3%** utilization of the NPU's peak throughput. In addition, despite using a batch size of 6, we still achieve very low latency. The biggest GRU workload with a very long sequence length of 256 takes only **1.1 ms**. To put this into context, the average sentence length in the English language is 15-20 words [40], which means that we can run inference over $11,600$-$15,500$ sentences per second. This shows that while we use batch 6 (instead of batch 1) for inference, we still meet real-time application requirements, while achieving higher throughput and hardware utilization.

Fig. 5 shows the performance of the three NPU versions under study across a diverse set of workloads. We normalize the results

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

	Nvidia V100†	Nvidia T4†	Intel S10 NX‡
Peak FP32 TOPS	15.7	8.1	3.96
Peak FP16 TOPS	(125)	(65)	143*
Peak INT8 TOPS	62.8	(130)	143
On-chip Mem. (MB)⋆	16	10	16
Process Tech.	TSMC 12nm	TSMC 12nm	Intel 14nm
Die Size (mm^2)	815	545	$<$ 500 [43]

† Perf. in brackets is with tensor cores ‡ FPGA peak perf. at 600 MHz
* Using block floating point ⋆ Register Files for GPUs, M20Ks for FPGA

by the number of logic elements in each device to account for the difference in device sizes. Assuming equal hardware utilization across devices for a given workload, the maximum performance gain that could possibly be achieved by our enhanced NPU can be computed from the peak TOPS/LE of Table I: 5.4× and 5.1× vs. the baseline NPU on the MX and GX devices, respectively. Fig. 5 shows that our enhanced NPU achieves gains very close to the maximum possible gains on bigger workloads (e.g. 5.2× and 5.1× on LSTM-1024, and 5.4× and 5.1× on RNN-1792). The geomean speedup over the baseline NPU on MX and GX across all studied workloads is ∼3.5×, due to the lower utilization of our enhanced NPU on smaller workloads.

V. Core Compute Benchmarking

A. Experimental Setup

In this section, we compare the performance of our enhanced NPU on the Stratix 10 NX AI-optimized FPGA against the latest accessible AI-optimized GPUs from Nvidia, the T4 and V100 . The T4 and V100 have 320 and 640 tensor cores (specialized matrix multiplication engines for AI workloads) respectively [41], [42]. Table III summarizes key specifications for these GPUs compared to Stratix 10 NX. The three devices use similar generation process technology. The V100 is the largest Nvidia 12nm GPU and is almost 50% bigger than the T4, while Stratix 10 NX is smaller than both GPUs [43].

First, we perform GPU micro-benchmarking using the GPU's favourite workload: square matrix-matrix multiplication (GEMM) to ensure that our setup is optimal by reproducing similar GEMM performance as in prior studies [44]. To measure the best GPU performance, we use the latest library for each device and precision that does not error out, and measure the performance with and without the tensor cores enabled. For the `fp32` and `fp16` experiments, we use the cuBLAS library from CUDA 10.0 and 10.2 for V100 and T4, respectively. For `int8`, we use the cuBLASLt library from CUDA 10.2 which achieves higher `int8` performance than cuBLAS [44]. We use Nvidia's official (highly-optimized) cuDNN kernels for the DL workloads. We use cuDNN 7.6.2 and 7.6.5 for the V100 and T4, respectively. The cuDNN libraries do not support `int8` compute kernels; however, they support a persistent mode which, if possible, keeps all the model weights in on-chip memory for higher performance similarly to the NPU persistent approach. For every workload, problem size and sequence length, we exhaustively run all possible configurations in terms of precision {`fp32, fp16, int8`}, compute style {persistent, non-persistent}, tensor core settings {enabled, disabled} on both GPUs. Then, we pick the best achieved performance to compare to the NPU on Stratix 10 NX. All the results in this section are measured using the `cudaEventRecord()` API which records time stamps on the GPU device at the specified points. Therefore, we only account for the core computation excluding any initialization, kernel launch or host-GPU data transfer overheads, which corresponds exactly to the NPU execution cycles on the FPGA. In Section VI, we include all these overheads for the GPU and FPGA to perform an end-to-end system-level performance evaluation.

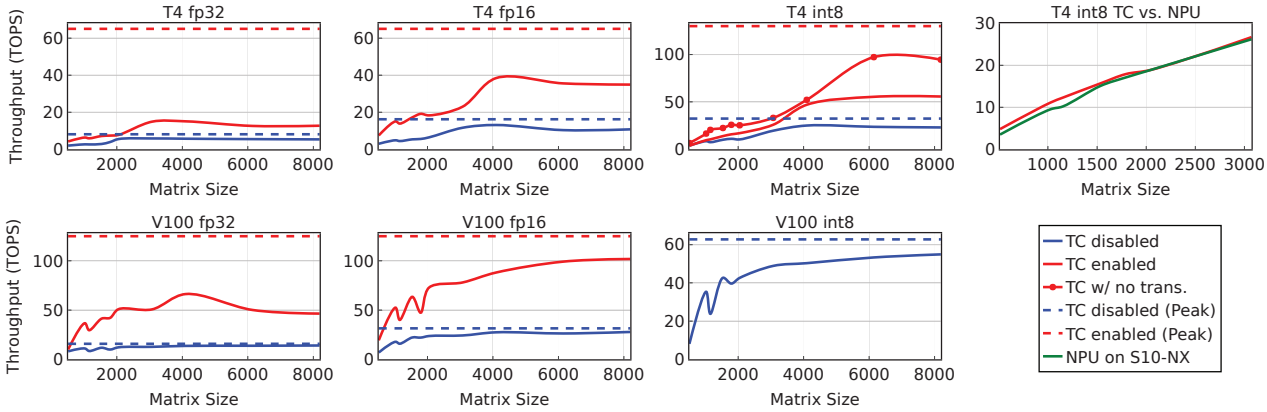

Fig. 6: Achievable and peak GEMM performance for T4 and V100 GPUs at different precisions, and tensor core (TC) settings. The V100 TCs do not support `int8`. Top right plot compares T4 vs. NPU GEMM performance for matrix sizes that can be kept persistent in NPU.

B. Know The Competition: GEMM Micro-Benchmarking

Fig. 6 shows the GEMM micro-benchmarking results for `fp32`, `fp16` and `int8` precisions on the T4 and V100 GPUs. The results show that tensor cores can significantly increase the GPUs' performance on GEMM (red lines) vs. the tensor cores disabled case (blue lines). However, a general trend is that the tensor cores, despite being designed for GEMM, are still significantly underutilized compared to their peak performance (red dashed lines) at matrix sizes of 2048 or below. They do not achieve high utilization except at very large matrix sizes that are uncommon in real DL workloads, which is similar to prior findings in [44]. The tensor cores on both the T4 and V100 do not support `fp32` precision; instead `fp32` data is converted into `fp16` before executing the multiplication operations on the tensor cores [45]. This data conversion overhead decreases tensor cores performance vs. pure `fp16` GEMMs. Another interesting observation is that when the T4 tensor cores are operating in `int8` mode, they require transforming the input matrices from the standard row/column major formats to a special tensor-core-specific layout [46]. As a result, the achieved `int8` performance on tensor cores (red line without markers) is less than 45% of the peak performance, even when processing very large 8192×8192 matrices. To better understand the overhead of this transformation, we conduct an additional experiment in which we supply the input matrices in the tensor core special layout (red line with markers). Even without the matrix layout transformation overhead, tensor core utilization is less than 40% for sizes of 4096×4096 and below, with a maximum of 72% utilization at 6144×6144 matrices.

We implement GEMM on our NPU by keeping one matrix persistent on-chip and streaming in the other matrix as a sequence of row vectors. The top right plot of Fig. 6 compares the NPU performance on Stratix 10 NX to the T4 GPU with `int8` tensor cores. For a fair comparison, we disable the matrix layout transformation for one of the two input matrices which corresponds to the persistent matrix on the NPU side. However, we keep the layout transformation for the second input and output matrices, since the NPU consumes and produces these matrices in standard formats. Although the NPU was designed for matrix-vector operations, it still achieves similar performance to T4 on GEMM workloads with sizes ranging from 512 to 3072 (the biggest matrix that can fit persistently in on-chip BRAMs). We could exploit the FPGA's reconfigurability to customize the NPU overlay for GEMM by adding a systolic matrix multiplication unit that maps well to the NX tensor blocks. However, that is out of scope for this work, since target workloads studied here do not use large GEMMs.

C. Performance Comparison on AI Workloads

Fig. 7 compares the performance of our enhanced NPU on Stratix 10 NX to the best T4 and V100 performance, across the exhaustive combination of configurations detailed in Section V-A, on GEMV and real-time DL workloads. The NPU performance is always significantly higher than both GPUs for small batches of size 3 and 6. The NPU performs best at batch-6 (which it was designed for: 2 cores at batch-3 each) with a $\mathbf{24.2}\times$ and $\mathbf{11.7}\times$ higher performance on average compared to the T4 and V100, respectively. Our NPU is less performant at batch-3 compared to batch-6 since one of the two cores is completely idle. However, it still achieves average speedups of $\mathbf{22.3}\times$ and $\mathbf{9.3}\times$ compared to the T4 and V100, respectively. At batch sizes higher than 6, the NPU can be underutilized if the batch size is not divisible by 6. For example, at batch sizes of 8, 32 and 256, the NPU can achieve at most 67%, 89% and 99% of its batch-6 performance respectively, while batch sizes of 12, 36 and 258 (shown as dashed lines in Figure 7) would all achieve 100% efficiency. At medium size batch with 32 inputs, the NPU still has better performance than the T4, and better or comparable performance to the V100. Even at large batches of size 256, our NPU has 58% higher performance than the T4, and only 30% less performance than the more powerful and larger V100. These results show that AI-optimized FPGAs can not only achieve an order of magnitude better performance than GPUs at low-batch real-time inference, but also compete in high-batch inference with relaxed latency constraints. The bottom right plot in Fig. 7 summarizes the geomean speedups of all studied workloads relative to the T4 performance at different batch sizes.

The top right plot of Fig. 7 shows the geomean utilization of the NPU compared to both GPUs at different batch sizes. The NPU achieves a geomean utilization of 37.1% at batch-6 compared to 1.5% and 3% for the T4 and V100, respectively. GPU tensor cores are not directly connected to each other [47]; they can only receive inputs from local in-core register files. Therefore, each GPU tensor core has to send its partial result (e.g. subset of output vector in RNN) to global memory and synchronize with other tensor cores to combine these partial results. The GPU then reads the combined vector from global memory to perform further operations, such as activation functions. A higher batch size can amortize such synchronization latency, but even at batch-256, the GPU utilization is only 13.3% and 17.8% in the T4 and V100, respectively. On the other hand, the FPGA has dedicated interconnect between the tensor blocks for reduction. The FPGA's programmable routing also allows cascading the MVU tiles and the vector elementwise engines for direct spatial communication, alleviating the need to communicate through memory as the case in the GPUs.

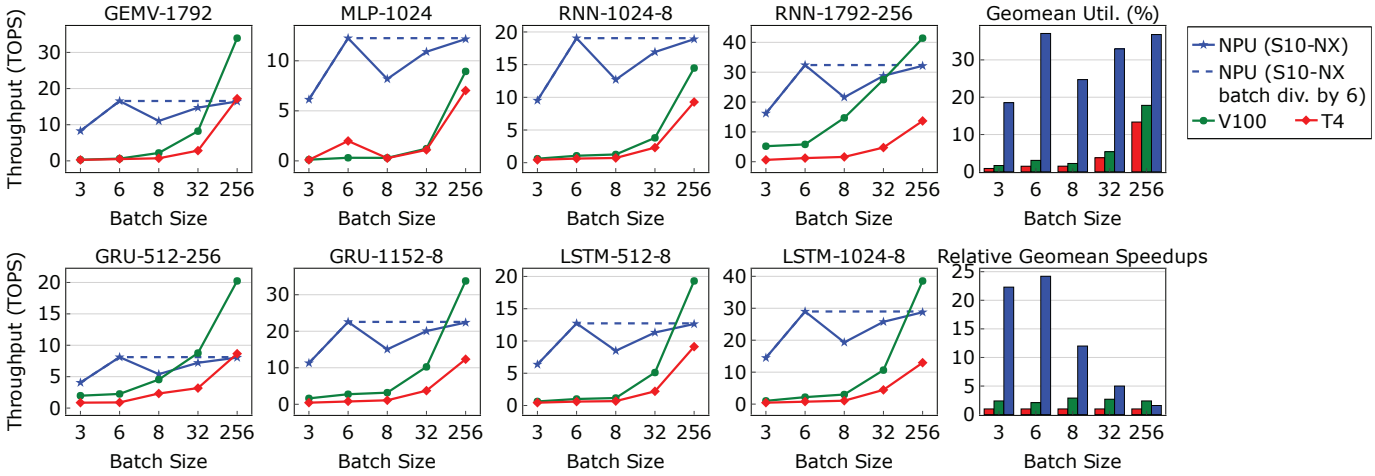

Fig. 7: Performance of V100 and T4 vs. NPU on the Stratix 10 NX FPGA for a portion of our DL benchmark suite at different batch sizes: 3, 6, 8, 32, 256. Dashed lines show NPU performance for larger batch sizes that are divisible by 6. NPU achieves 24.2× and 1.6× speedups over T4 for batch 6 and 256 across all workloads (bottom right plot).

TABLE IV: Specifications summary for the GPU and FPGA systems.

	CPU Host	OS	Host-Device Link	BW (Gbps)
Nvidia T4 (AWS)	Intel Xeon Platinum 8259CL @2.5 GHz	Virt. Linux	PCIe Gen3x16	128
Nvidia V100 (TACC)	Intel Xeon Platinum 8160 @2.1 GHz	Linux	PCIe Gen3x16	128
Intel S10 MX/NX	Intel i7-4790 CPU @3.6GHz	Linux	PCIe Gen3x16* 100G Ethernet†	128 100

* Between client CPU and its network interface card (NIC)
† Between client NIC and FPGA

VI. System-Level Benchmarking

A. GPU & FPGA Systems for AI Acceleration

A typical GPU-based inference system consists of a server with the host CPU connected to a GPU card via a PCIe interface. An inference request sent from a remote client goes first to the server's NIC, then to host CPU, and finally to the GPU card. For the FPGA system, we leverage the tightly integrated Ethernet interface on the FPGA to directly receive inputs from a remote client similarly to Microsoft's Brainwave [25]. To get concrete insights on real end-to-end system performance, we evaluate the aforementioned prototypical GPU and FPGA systems, as detailed in Table IV. We have physical access to the V100 GPU on TACC [49], and we access the T4 GPU on an AWS virtual machine instance [50].

B. Steps of an End-to-End AI Inference Application

An end-to-end AI inference workload goes beyond just core compute on the accelerator card, as Table V highlights. Generally, it consists of one-time initializations, preparing and sending application input data to the accelerator, accelerator execution, and sending back results. For the FPGA-based system, the remote client CPU needs to access its NIC with optimized software libraries. We use the Data Plane Development Kit (DPDK) [48] in this study. On the other hand, we use Nvidia-recommended methods for interaction between the host CPU and GPU card, such as pinning the host memory space to be transferred to the GPU device using `cudaMallocHost()` to avoid paging overhead. Stage 1 only needs to be performed once for any number of inference requests, and stage 2 is common for both the GPU and FPGA systems. Therefore, we focus only on stages 3-5 in our study. For a fair comparison, since the NPU weights in the FPGA system are persistently stored on-chip, we do not include the `cudaMemcpy()` for weights in our evaluation.

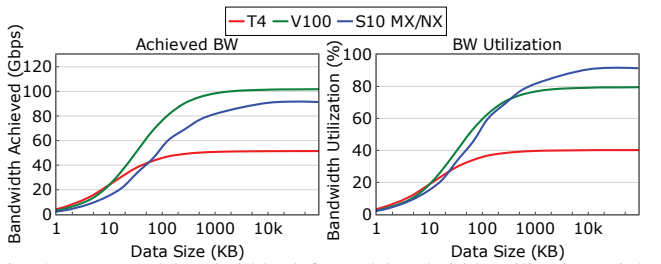

Fig. 8: Measured bandwidth (left) and bandwidth utilization (right) on Ethernet-based FPGA and PCIe-based GPU systems.

C. Data Movement Efficiency Characterization

We first study the efficiency of moving data to/from the FPGA via ethernet and the GPUs via PCIe. A loopback logic is implemented on the FPGA using Intel's 100G Ethernet Hard IP [51], which immediately sends the received packets back to the CPU. For our experiments in this section, we use a network-connected Stratix 10 MX board, which has the exact same 100G Ethernet interface as the NX board, to evaluate the FPGA system overheads. On all systems, we measure host-to-device and device-to-host data transfer performances separately and take their average. As shown in Fig. 8, the measured bandwidth on the 3 systems first increases with the data size and then it saturates. The V100 GPU system achieves the highest bandwidth at all data sizes, since its PCIe offers the highest peak bandwidth. We observe that the T4 system realizes lower bandwidth, possibly due to the AWS virtualization overheads (e.g. when sending GPU input data from a memory space pinned in the virtual machine's physical memory). The FPGA achieves the best utilization of its 100G Ethernet interface with up to 90% utilization, whereas the V100 can only utilize up to 80% of its peak 128 Gbps PCIe bandwidth.

D. End-to-end Performance Comparison on AI Workloads

Fig. 9 shows system-level execution time of RNN workloads at batch-6 and sequence lengths of 8 (short) and 256 (long). After including the system overheads, the FPGA system achieves 16-19× and 15-25× speedups on RNNs with short input sequences and 11-16× and 5-6× for long sequences compared to the T4 and V100 systems, respectively. These speedups are less than the core compute speedups reported in Section V due to Amdahl's law; the FPGA system does not achieve the same speedup for system overhead as for kernel execution time compared to the GPU systems. For long sequences with more output data, the NPU achieves ∼2× lower system overhead, since it overlaps streaming

TABLE V: End-to-end steps of an AI inference workload on both the FPGA and GPU systems.

#	Stage	FPGA System	GPU System
1	**One-time Initialization**	Allocate memory on remote CPU (e.g. using DPDK [48] to allocate NIC packet buffers).	Allocate memory on local CPU host and GPU device using `cudaMallocHost()` and `cudaMalloc()`.
2	**Prepare inputs**	Initialize remote CPU host memory with input data (e.g. in the DPDK TX buffer).	Initialize local CPU host memory with input data.
3	**Send input to accelerator**	DPDK APIs are used to construct the Ethernet packets and send them via the NIC. For RNN workloads, hidden vectors initialization is done on the NPU.	Move data from host memory to GPU memory using `cudaMemcpy()`. For RNN workloads, host calls a GPU kernel to initialize hidden vectors.
4	**Accelerator execution**	FPGA receives inputs from Ethernet, and triggers NPU execution. NPU execution is non-blocking and can send back partial results (one time step in RNN) once ready.	Call GPU application library (e.g. RNN in cuDNN). Execution is blocking and can send final results (i.e. all time steps in RNN) only after GPU execution is finished.
5	**Send back results**	Remote CPU host receives results from its NIC. We use DPDK to access NIC buffers.	Copy result from GPU memory back to host using `cudaMemcpy()`.

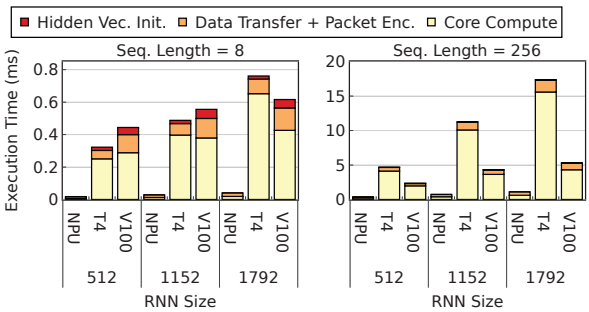

Fig. 9: System-level execution time of RNN workloads for short (left) and long (right) sequences.

out the per-step result with the computation of subsequent steps. In contrast, the GPU's predefined programming model requires that it computes results of all time steps before they are accessed by the CPU host. In addition, the GPUs require a kernel call to initialize their states (such as the hidden vector in RNNs) [52], while the FPGA's flexible programming model does not suffer from such overheads. For short sequences, such overheads become prominent as data transfer and kernel execution times are reduced. Hence, the FPGA system overhead in this case is even smaller; $\sim$5$\times$ and $\sim$10$\times$ better than the T4 and V100, respectively.

VII. Power Analysis

For the GPUs, power is measured with the Nvidia SMI tool, which reports power for the entire GPU card (including the packaged GPU chip and other components such as memories and fans), as well as the GPU die temperature [53]. Since the GPUs we use have aggressive cooling solutions (e.g. production-grade cooling in AWS and immersion cooling in TACC [54]), the Nvidia SMI tool reports 35°C and 40°C core GPU temperature for the V100 and T4, respectively. For all the workloads we study, the GPU is significantly underutilized, and therefore the measured GPU power is in the range of 27-45W for the T4 and 35-72W for the V100. However, when running a workload that achieves high utilization on the GPUs, such as 8192×8192 GEMM, the power consumption goes up to 70W and 190W for the T4 and V100, respectively.

We also measure the power of the Stratix 10 NX development kit using a high resolution power meter. The development kit we use has an air-cooled heat sink and on-board temperature sensors. We carry out all our experiments with room temperature ambient and the FPGA board sitting on a desk in our lab without any special cooling solutions. We configure the NPU to run GEMV and DL workloads in an infinite loop and record the power and temperature measurements after a few minutes of continuously running each workload. The measured power is in the range of 54-70W at measured board temperatures of 35-39°C depending on the NPU utilization of the running workload (as indicated in Table II). These results show that the Stratix 10 NX NPU running batch-6 inference achieves 12-16$\times$ and 8-12$\times$ higher average energy efficiency (i.e. TOPS/Watt) on the studied workloads compared to the T4 and V100 GPUs, respectively.

VIII. Related Work

We discussed related work on the baseline NPU overlay and various FPGA architecture optimizations for DL in Sections II and III, respectively. In this paper, we evaluated the performance of the new Stratix 10 NX FPGA on different variations of RNN models. Several prior works have focused on accelerating RNNs on FPGAs [5], [13], [55]–[59]. Our enhanced NPU utilizing the NX tensor blocks outperforms all prior work on FPGA-based RNN acceleration, even the ones that use lower than `int8` precision, on the studied workloads. Some prior studies also compared the performance of FPGAs and GPUs on AI workloads such as [60]–[62]. However, this work is the first performance evaluation of an AI-optimized FPGA, the Stratix 10 NX, with integrated tensor blocks in comparison to the latest accessible AI-optimized GPUs with tensor cores. We show that at comparable peak TOPS, the FPGA's flexible architecture can offer significantly higher utilization of tensor units than GPUs. In addition, prior work has also studied host to FPGA PCIe communication overheads [15], [63], as well as GPU system overheads [64]. Our work studies 100G Ethernet-connected FPGA system overheads in comparison to PCIe-connected GPUs, especially for real-time DL workloads.

IX. Conclusion

In this work, we presented the first evaluation of the performance of Intel's AI-optimized Stratix 10 NX FPGA with tensor blocks compared to the latest accessible AI-optimized Nvidia GPUs, the T4 and V100. We proposed enhancements to the prior Brainwave NPU architecture, ISA, and toolchain to restructure computation to best leverage the tensor blocks in NX. Our enhanced NPU on Stratix 10 NX achieves 3.5$\times$ higher performance per LE than the baseline NPU (without tensor blocks) on Stratix 10 MX/GX on average across key AI workloads, with up to 80% NPU utilization. On the other hand, we show that even for simple square GEMM benchmarks, the GPU tensor cores can be significantly underutilized. Our enhanced NPU on Stratix 10 NX delivers 24$\times$ and 12$\times$ higher core compute performance on average compared to the T4 and V100 GPUs at batch-6, despite the smaller NX die size. Even at a large batch size of 256, our NPU still achieves 58% higher performance than the T4, and only 30% less performance than the larger V100. Finally, we evaluate the (commonly overlooked) end-to-end system level overheads in both FPGA-based and GPU-based AI inference systems. We show that the FPGA's integrated 100G Ethernet results in 10$\times$ and 2$\times$ less overhead compared to the 128 Gbps PCIe interface on the V100 GPU for RNN workloads with short and long input sequences. Including all system-level overheads, the NPU on Stratix 10 NX averages approximately an order of magnitude speedup compared to the studied GPUs.

REFERENCES

[1] I. Akkaya *et al.*, "Solving Rubik's Cube with a Robot Hand," *arXiv preprint arXiv:1910.07113*, 2019.

[2] A. Radford *et al.*, "Language Models are Unsupervised Multitask Learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[3] D. Silver *et al.*, "Mastering the Game of Go Without Human Knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[4] C. Berner *et al.*, "DOTA 2 with Large Scale Deep Reinforcement Learning," *arXiv preprint arXiv:1912.06680*, 2019.

[5] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[6] N. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[7] K. Hazelwood *et al.*, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[8] Dale Southard, "Tensor Streaming Architecture Delivers Unmatched Performance for Compute Intensive Workloads," 2019.

[9] Z. Jia *et al.*, "Dissecting the Graphcore IPU Architecture via Microbenchmarking," *arXiv preprint arXiv:1912.03413*, 2019.

[10] D. Abts *et al.*, "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads," in *International Symposium on Computer Architecture (ISCA)*, 2020, pp. 145–158.

[11] B. Gaide *et al.*, "Xilinx Adaptive Compute Acceleration Platform: Versal Architecture," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 84–93.

[12] E. Nurvitadhi *et al.*, "In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 106–1064.

[13] ——, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 199–207.

[14] Intel Corp., "Intel Stratix 10 NX FPGA: AI-Optimized FPGA for High-Bandwidth, Low-Latency AI Acceleration (SS-1121-2.0)," 2020.

[15] E. Nurvitadhi *et al.*, "Scalable Low-Latency Persistent Neural Machine Translation on CPU Server with Multiple FPGAs," in *International Conference on Field-Programmable Technology (FPT)*, 2019, pp. 307–310.

[16] M. Jain *et al.*, "RNN-T for Latency Controlled ASR with Improved Beam Search," *arXiv preprint arXiv:1911.01629*, 2019.

[17] T. Hastie *et al.*, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, 2009.

[18] M. Naumov *et al.*, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *arXiv preprint arXiv:1906.00091*, 2019.

[19] I. Goodfellow *et al.*, *Deep Learning*. MIT press, 2016.

[20] K. Cho *et al.*, "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation," *arXiv preprint arXiv:1406.1078*, 2014.

[21] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[22] V. J. Reddi *et al.*, "MLPerf Inference Benchmark," in *International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[23] *DeepBench*, (accessed August 5, 2020). [Online]. Available: https://github.com/baidu-research/DeepBench

[24] F. Zhu *et al.*, "Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-chip," *arXiv preprint arXiv:1804.10223*, 2018.

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

[26] J. Fowers *et al.*, "Inside Project Brainwave's Cloud-Scale, Real-Time AI Processor," *IEEE Micro*, vol. 39, no. 3, pp. 20–28, 2019.

[27] A. Boutros *et al.*, "Math Doesn't Have to Be Hard: Logic Block Architectures to Enhance Low-Precision Multiply-Accumulate on FPGAs," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 94–103.

[28] M. Eldafrawy *et al.*, "FPGA Logic Block Architectures for Efficient Deep Learning Inference," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 3, pp. 1–34, 2020.

[29] S. Rasoulinezhad *et al.*, "LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 161–171.

[30] 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, pp. 35–357.

[31] Intel Corp., "Intel Agilex Variable Precision DSP Blocks User Guide (UG-20213)," 2020.

[32] S. Rasoulinezhad *et al.*, "PIR-DSP: An FPGA DSP Block Architecture for Multi-Precision Deep Neural Networks," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 35–44.

[33] Achronix Semiconductor Corp., "Speedster7t Machine Learning Processing User Guide (UG088)."

[34] A. Arora *et al.*, "Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 53–60.

[35] L. Gwennap, *Stratix 10 NX Adds AI Blocks*, 2020 (accessed August 5, 2020). [Online]. Available: https://www.linleygroup.com/newsletters/newsletter_detail.php?num=6183&year=2020&tag=3

[36] Intel Corp., "Intel Stratix 10 Variable Precision DSP Blocks User Guide (UG-S10-DSP)," 2020.

[37] A. Boutros *et al.*, "You Cannot Improve What You Do Not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[38] M. Langhammer *et al.*, "Fractal Synthesis: Invited tutorial," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 202–211.

[39] M. Langhammer and other, "Extracting INT8 Multipliers From INT18 Multipliers," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 114–120.

[40] A. Moore, "The Long Sentence: A Disservice to Science in the Internet Age," *BioEssays*, vol. 33, no. 12, pp. 193–193, 2011.

[41] Nvidia Corp., "Nvidia T4 Tensor Core GPU (Device Datasheet)," 2019.

[42] ——, "Nvidia Tesla V100 GPU Architecture (WP-08608-001)," 2017.

[43] *Private communication with Intel*.

[44] Z. Jia *et al.*, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[45] *Nvidia Deep Learning SDK Documentation*, (accessed August 5, 2020). [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/cudnn-archived/cudnn_765/cudnn-api/index.html#cudnnSetRNNMatrixMathType

[46] *Nvidia CUDA Toolkit Documentation*, (accessed August 5, 2020). [Online]. Available: https://docs.nvidia.com/cuda/archive/10.2/cublas/index.html#cublasLt-example-tensorop

[47] Nvidia Corp., "Nvida Turing GPU Architecture White Paper (WP-09183-001_v01)."

[48] *Data Plane Development Kit (DPDK)*, (accessed August 5, 2020). [Online]. Available: https://www.dpdk.org/

[49] T. U. of Texas at Austin, *Texas Advanced Computing Center*. [Online]. Available: https://www.tacc.utexas.edu/

[50] Amazon, *Amazon EC2 G4 Instances*. [Online]. Available: https://aws.amazon.com/ec2/instance-types/g4/

[51] Intel Corp., "Intel Stratix 10 H-Tile Hard IP for Ethernet IP Core User Guide (UG-20121)," 2018.

[52] *CUDA C++ Programming Guide*, (accessed August 5, 2020). [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[53] *NVIDIA System Management Interface*, (accessed August 5, 2020). [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[54] *The Maverick2 Cooling System in TACC*. [Online]. Available: https://portal.tacc.utexas.edu/documents/10157/1181317/Maverick2+cooling+system/fbef2f24-4252-4d5f-857e-73c138ff6a0e?t=1592320888902

[55] Y. Guan *et al.*, "FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 629–634.

[56] S. Han *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 75–84.

[57] V. Rybalkin *et al.*, "FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 89–897.
[58] Z. Que *et al.*, "Optimizing Reconfigurable Recurrent Neural Networks," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 10–18.
[59] V. Rybalkin *et al.*, "Efficient Hardware Architectures for 1D-and MD-LSTM Networks," *Journal of Signal Processing Systems*, pp. 1–27, 2020.
[60] E. Nurvitadhi *et al.*, "Evaluating and Enhancing Intel Stratix 10 FPGAs for Persistent Real-Time AI," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 119–119.
[61] ——, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 5–14.
[62] ——, "Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.
[63] D. J. Moss *et al.*, "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 107–116.
[64] D. Lustig and M. Martonosi, "Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 354–365.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/benchmarks.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Tests measure performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

♻ Please Recycle