2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs

Eriko Nurvitadhi, Dongup Kwon, Ali Jafari, Andrew Boutros, Jaewoong Sim, Phillip Tomson, Huseyin Sumbul, Gregory Chen, Phil Knag, Raghavan Kumar, Ram Krishnamurthy, Sergey Gribok, Bogdan Pasca, Martin Langhammer, Debbie Marr and Aravind Dasu
Intel Corporation
E-mail: eriko.nurvitadhi@intel.com

***Abstract*—Interactive intelligent services, such as smart web search, are important datacenter workloads. They rely on data-intensive deep learning (DL) algorithms with strict latency constraints and thus require balancing both data movement and compute capabilities. As such, a persistent approach that keeps the entire DL model on-chip is becoming the new norm for real-time services to avoid the expensive off-chip memory accesses. This approach is adopted in Microsoft's Brainwave and is also provided by Nvidia's cuDNN libraries. This paper presents a comparative study of FPGA, GPU, and FPGA+ASIC in-package solutions for persistent DL. Unlike prior work, we offer a fair and direct comparison targeting common numerical precisions (FP32, INT8) and modern high-end FPGA (Intel® Stratix®10), GPU (Nvidia Volta), and ASIC (10 nm process), all using the persistent approach. We show that Stratix 10 FPGAs offer 2.7× (FP32) to 8.6× (INT8) lower latency than Volta GPUs across RNN, GRU, and LSTM workloads from DeepBench. The GPU can only utilize ∼6% of its peak TOPS, while the FPGA with a more balanced on-chip memory and compute can achieve much higher utilization (∼57%). We also study integrating an ASIC chiplet, TensorRAM, with an FPGA as system-in-package to enhance on-chip memory capacity and bandwidth, and provide compute throughput matching the required bandwidth. We show that a small 32 mm^2 TensorRAM 10nm chiplet can offer 64 MB memory, 32 TB/s on-chiplet bandwidth, and 64 TOPS (INT8). A small Stratix 10 FPGA with a TensorRAM (INT8) offers 15.9× better latency than GPU (FP32) and 34× higher energy efficiency. It has 2× aggregate on-chip memory capacity compared to a large FPGA or GPU. Overall, our study shows that the FPGA is better than the GPU for persistent DL, and when integrated with an ASIC chiplet, it can offer a more compelling solution.**

I. Introduction

With deep learning (DL) algorithms becoming the backbone of various real-time datacenter services, there is a huge demand for acceleration solutions that can meet the strict latency constraints and limited power budget of such services. Despite their large peak throughput numbers, GPUs cannot utilize such compute throughput when there is limited compute intensity (low ops/byte), such as in the case of real-time low-batch low-latency DL applications. ASICs can deliver the highest energy efficiency at the cost of fixed functionality and high non-recurring engineering (NRE) cost, which poses a huge challenge in the DL domain with continuously changing algorithms and precisions. On the other hand, FPGAs offer the advantages of reconfigurability, flexible memory hierarchy, and less NRE cost and time-to-market, at the cost of 9× bigger and 3-6× slower DL accelerators compared to ASIC ones [1].

Although most of the recent work on hardware acceleration of DL focuses on convolutional neural networks (CNNs), it was shown that CNNs represent only 5% of datacenter DL workloads [2]. Trending datacenter services rely on data-intensive models such as recurrent neural networks (RNNs), gated recurrent units (GRUs), long short-term memories (LSTMs), and multi-layer perceptrons (MLPs). These models suffer from memory bottlenecks due to their low compute-to-data ratio, unlike the extensively studied compute-bound CNNs. For these workloads, performance depends not only on the computational throughput, but also on efficiently moving data through the memory system to/from the compute units.

Since accessing off-chip memory is an order of magnitude more expensive than on-chip accesses [3], a new persistent approach that keeps the entire model on-chip is becoming the new norm for real-time DL acceleration. For instance, Microsoft's Brainwave (BW) stores the model parameters in the on-chip BRAMs of one or more FPGAs. Also, the latest Nvidia cuDNN library supports persistent mode that stores the model in the GPU's register files and caches. Both a high-end Intel Stratix 10 FPGA and an Nvidia Titan V GPU contain ∼30 MBs of on-chip memory, which is likely to grow in the future providing a stronger case for persistent DL processing.

Given similar on-chip memory capabilities and support for persistent approaches in the latest high-end FPGAs and GPUs, it is critical to evaluate which platform can perform better on persistent DL. To our knowledge, such direct benchmarking has not been done before. Prior work [4] reported persistent FPGA results, but in comparison to non-persistent results of a GPU that is one generation older and uses a different precision. Other prior work report results only for persistent DL on GPUs [5], [6] or FPGAs [7]. In this paper, we present a fair and direct comparison of the latest Stratix 10 FPGAs and Volta GPUs in the context of real-time persistent DL sequence models such as RNNs, GRUs and LSTMs. We then make a case for FPGA-ASIC integration by leveraging the system-in-package (SiP) nature of Intel Stratix 10 FPGAs. Recent work has shown that FPGA-ASIC integration is promising for compute-intensive DL algorithms such as CNNs [8]. However, in this work, we focus on a different, and arguably more prevalent, class of data-intensive applications with considerably lower compute-to-data ratio. Our contributions are the following:

- We implement an optimized BW-like neural processing unit (NPU) in RTL and a complete NPU development stack, to serve as a state-of-the-art benchmark for our study.
- We present a direct comparison of our NPU on a high-end Stratix 10 FPGA and an equivalent high-end Volta GPU running persistent kernels using common precisions (FP32, INT8/4) across a wide variety of workloads.
- We propose TensorRAM, a new 32-mm^2 10nm Stratix 10 chiplet for persistent data-intensive DL sequence models that offers 64 MB of on-chiplet SRAMs with matching near-memory variable-precision compute throughput.

2576-2621/19/$31.00 ©2019 IEEE
DOI 10.1109/FCCM.2019.00035

II. Background

A. Trends in Real-Time Intelligent Datacenter Services

In recent years, major advances in DL, along with the rapid growth of cloud computing and smart devices, have led to the proliferation of large-scale real-time intelligent services. For example, modern web search engines allow users to pose a question rather than provide simple keywords. Intelligent personal assistants are another example where real-time DL is used to process user speech and give smart responses.

Such services are latency critical because its response time directly relates to user experience. In 2014, Google reported that the latency requirement for their AI cloud-based inference was 10 ms, which decreased to 7 ms in 2016 [2]. While one common technique to improve compute efficiency is grouping multiple inputs together (i.e. batching), waiting for enough inputs to batch does not usually meet latency constraints. Microsoft reported that their BW AI cloud targets the lowest possible latency, processing a single input at a time [4].

B. Trends in DL Hardware Acceleration

The focus of to-date DL hardware accelerators, from general-purpose GPUs to specialized accelerators, has been offering *peak* compute throughput. This is because, not until recently, their main focus has been CNN workloads that have a high comptue intensity and rely on matrix-matrix multiplication operations. In addition, reporting impressive peak TOPS is a great marketing message for DL acceleration solutions.

However, real-world inference workloads with their low computational intensity and their strict latency demands are a mismatch to the claimed efficiency of these accelerators, which often utilize $<$10% of their peak TOPS. In this paper, we show that high-performance acceleration of real-time DL requires balancing memory bandwidth, storage, compute, and data movement at a different optimization point.

C. DL Sequence Models: RNNs, GRUs and LSTMs

RNNs are a class of neural networks that contain recurrent connections in the network. They achieve unprecedented accuracy in many sequence-based problems such as text analytics. There are many RNN variants, but the most popular ones are GRUs and LSTMs. LSTMs are composed of a number of memory cells, each of which is composed of three multiplicative gating connections called input, forget, and output gates. LSTMs use the following equations to compute the gates and produce the results for the next time step.

$$i_t = \sigma(x_t W_i + h_{t-1} U_i + b_i) \qquad f_t = \sigma(x_t W_f + h_{t-1} U_f + b_f)$$
$$o_t = \sigma(x_t W_o + h_{t-1} U_o + b_o) \qquad g_t = \tanh(x_t W_c + h_{t-1} U_c + b_c)$$
$$c_t = f_t \cdot c_{t-1} + i_t \cdot g_t \qquad h_t = o_t \cdot \tanh(c_t)$$

Here, i_t, f_t, o_t, c_t, h_t are the input gate, forget gate, output gate, cell state, and cell output at time step t. The $\cdot$ and $+$ operators denote an element-wise multiplication and an element-wise addition. σ and $\tanh$ are the sigmoid and hyperbolic tangent activation functions. W and U denote weight matrices, and the b terms denote bias vectors.

MLPs, which represent another class of common DL datacenter workloads [2], consist of back-to-back fully connected layers that can be formulated as matrix-vector multiplications. In this work, we decide to focus on the more complicated RNNs, GRUs and LSTMs. These models consist of multiple matrix-vector multiplications in addition to vector operations and recurrent connections. Thus, they represent a superset of the operations in the relatively simpler MLPs.

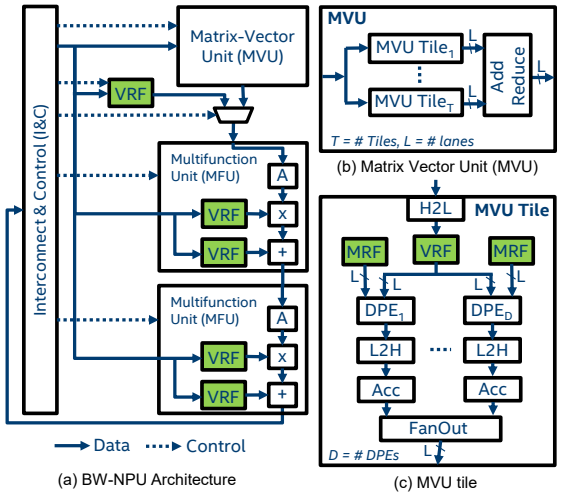

Fig. 1. BW-NPU, MVU, and MVU tile block diagrams.

III. The Brainwave NPU Architecture

There are various FPGA-based DL accelerators presented in literature such as [9], [7] and [10]. However, we chose Microsoft's Brainwave NPU (BW-NPU) as a benchmark for our study since it is commercially deployed and was designed specifically for persistent RNN, GRU, and LSTM workloads. We implement the BW-NPU from scratch in SystemVerilog based on the publicly available information [4] because we do not have access to the BW implementation from Microsoft.

A. Overview of the Brainwave NPU

The BW-NPU is a software-programmable FPGA overlay for persistent neural network inference. The NPU instruction set architecture (ISA) consists of compute instructions, such as matrix-vector multiplication and element-wise vector activations, addition, and multiplication, as well as instructions for data movement from and to matrix register files (MRFs) and vector register files (VRFs). Fig. 1a shows the block diagram of the BW-NPU architecture. It consists of a matrix-vector unit (MVU), two multifunction units (MFUs), in addition to interconnect and control (I&C). For brevity, we will include only the details that are necessary for understanding the rest of this paper. We recommend reviewing the details in [4] for a deeper understanding of the NPU architecture and ISA.

1) Matrix-Vector Unit (MVU): BW's MVU performs wide dot product operations between a shared input vector broadcasted from the VRFs and different matrix rows coming from the distributed MRFs in parallel. As depicted in Fig. 1b, the MVU consists of multiple tiles, connected to central reduction logic, which reduces partially computed vectors produced by the tiles into a final vector output of the MVU. An MVU tile contains a single VRF and D L-wide dot product engines (DPEs), each of which is coupled with an MRF. The DPEs perform the dot-product in low precision, then accumulate to a higher precision as indicated by the precision adjustment blocks (H2L and L2H) in Fig. 1c.

Fig. 2 illustrates the mapping of a matrix-vector multiplication operation to the MVU block for a matrix M of size 4×16

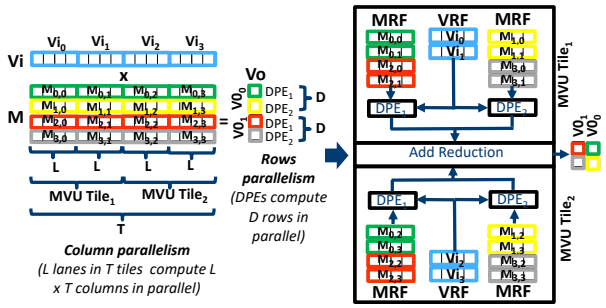

Fig. 2. An Example of mapping matrix-vector multiplication to MVU with 2 tiles, 2 DPEs and 4 lanes (i.e. 2T-2D-4L).

and a 16-element input vector V_i producing a 4-element output vector V_o. In this example, the MVU block has 2 tiles, 2 DPEs and 4 lanes. We denote such configuration as 2T-2D-4L and we will use the same notation for NPU configurations for the rest of this paper. The MVU takes advantage of all possible parallelism dimensions in a matrix-vector multiplication, by computing on multiple row blocks and column blocks in parallel. Each tile is responsible for a major column block of the matrix. Within the tile, each DPE operates on a distinct matrix row, while each lane in the DPE operates on a distinct element in that row. Reduction across lanes is performed inside the DPE to produce a partial result, which is then reduced across tiles to produce the final result.

2) Multi-Function Unit (MFU): The MFU block contains elementwise activation, high-precision multiplication, and add/subtract units. The activation unit implements different activation functions that are widely used in DNNs: tanh, sigmoid, and ReLU. We implemented the tanh and sigmoid cores using lookup tables of size 2048 similar to [11] while exploiting some of the properties of these functions such as symmetry and range-of-interest [12]. We verified that our approximate activation functions have a maximum error of only 9×10^{-4} and 5×10^{-4} compared to a full-precision software reference which is sufficient for our use case.

3) Interconnect & Control (I&C): The NPU is controlled using VLIW instructions. A top-level decoder translates each VLIW instruction into macro-operations (mOPs) that are dispatched to their respective modules. Each NPU module has a low-level decoder that produces a set of micro-operations (μOPs) to execute a single mOP. For example, an MVU mOP is a matrix-vector multiplication between a vector from a VRF address and a matrix from an MRF address. Such a mOP is translated into a sequence of μOPs to perform dot products, update accumulators, and write results to the output FIFO.

B. Hazard Detection & Resolution Mechanism

Data hazards in our NPU can happen when an instruction reads a vector operand that is produced by a previous instruction, resulting in a classic read-after-write hazard. Handling hazards is not discussed in [4], so we implemented our own hazard detection and resolution mechanism. In particular, since our instruction operates at a large granularity of operands and computations (i.e. vectors), we chose to implement hazards resolution by stalling.

A naïve stalling-based hazard resolution approach would be to stall the issue of an instruction, until the execution of all older instructions is complete. However, in our NPU programs, we found that this could lead to noticeable performance degradation. It is possible that the MVU already completes its mOP, but unnecessarily stalls as it cannot dispatch the next instruction's MVU mOP until the current instruction's MFU mOPs are finished. In our NPU, we chose a more optimized approach, where the NPU stalls only if it specifically detects a hazard between the mOPs of an instruction to be dispatched and those of an older instruction that has yet to be completed. This is done by checking the target VRF operand source and destination of the younger and older mOPs, respectively.

To run a benchmark on our NPU, we write an NPU program describing the benchmark as a sequence of matrix-vector and elementwise vector operations using the NPU ISA. This NPU program is then compiled using a compiler/assembler that we develop to automatically detect hazards between instructions and generate a binary that can be executed on the NPU. For the NPU programs implementing our benchmark workloads in this paper, the compilation time is less than 2 ms.

TABLE I
PERFORMANCE COMPARISON OF BW-NPU [4] VS. OUR NPU AFTER APPLYING A MORE FINE-GRAINED ZERO PADDING SCHEME.

	Latency (ms)		Speedup
	Our NPU	BW-NPU	
LSTM h=256 t=150	0.107	0.425	3.9
LSTM h=512 t=25	0.029	0.077	2.7
LSTM h=1024 t=25	0.040	0.074	1.8
LSTM h=1536 t=50	0.104	0.145	1.4
GRU h=1024 t=1500	2.337	3.792	1.6
GRU h=1536 t=375	0.734	0.951	1.3

C. Comparison Against BW-NPU

We developed our NPU such that it faithfully resembles the BW-NPU according to our interpretation of the publicly disclosed information in [4] except for two minor differences:

- We focus on common numerical precisions such as FP32 and INT8/4 instead of Microsoft's non-standard floating point formats (FP8/11). Using such precisions enables a conservative comparison to GPUs. If FPGAs already offer better performance than GPUs at these GPU-friendly precisoins, then there are more custom precisions that can be used by FPGAs (e.g. MS-FP8) to provide even better performance. We also use INT27 for accumulation and vector operations, since it is more friendly to the FPGA's hard DSP blocks.
- In the MFU, we choose to connect the activation, multiply, and add/subtract compute units in series instead of having a full crossbar as in BW. This is more resource-efficient and routing-friendly, but less flexible since certain combinations of operations become not possible. However, we did not measure a substantial impact for excluding it and were able to accommodate all compositions needed for our RNN, GRU, and LSTM benchmarks.

In order to make sure that our NPU architecture is comparable to BW-NPU, we ran RTL simulation with BW-NPU's HW configuration (i.e. 6T-400D-40L) for several GRU/LSTM benchmarks. The results showed that our NPU performs within 10% of BW-NPU results reported in [4]. We then optimized beyond BW-NPU by allowing a more fine-grained zero-padding scheme that depends on the number of lanes (L) instead of the number of DPEs (D). This optimization resulted in an average of 90% additional performance improvement for small and medium size workloads running on large NPU configurations such as 6T-400D-40L as shown in Table I.

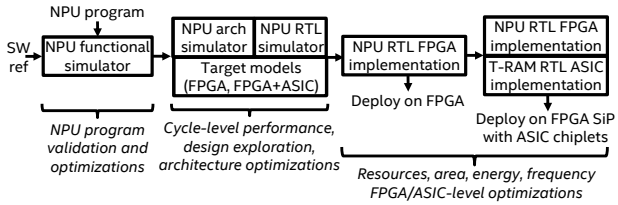

Fig. 3. Our complete NPU development flow.

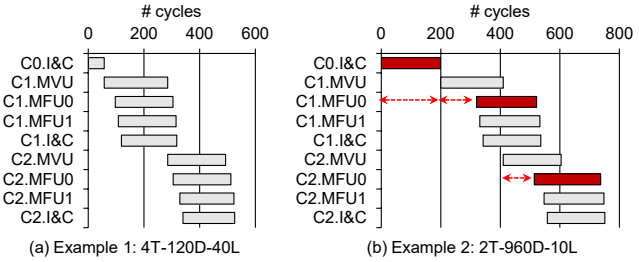

Fig. 4. Example simulation traces for our NPU with two different configurations for a single-step RNN workload.

IV. Our NPU Framework & FPGA Implementation

A. NPU Development Flow

We implemented a complete NPU development flow as shown in Fig. 3. First, we built an ISA-level NPU functional simulator (*FSim*) to run NPU programs. This is useful for validating an NPU program by comparing its functionality against a software reference. The FSim is usable by any DL application developer, without any expertise in FPGA design or hardware acceleration. The FSim also allowed us to experiment with traditional compiler optimizations, such as loop unrolling and instruction re-ordering to avoid dependencies. Second, to evaluate cycle-level performance, we built an NPU architecture performance simulator (*PSim*) in C++. Links, interfaces, memory/compute structures are parameterizable for rapid design space exploration of NPU architectures. For more precise cycle-level modeling, we also built parameterizable Verilog RTL testbenches. RTL simulation is much slower than PSim but more accurate. Nevertheless, we found that the PSim predicts runtime cycles within $\sim$10% accuracy of the RTL simulation. Finally, we apply FPGA-specific optimizations in the compute units and data distribution for higher performance. We built our performance simulators (PSim and RTL) to model both FPGA only and FPGA integrated with ASIC chiplets as SiP as we discuss in Section V.

B. Design Space Exploration

An NPU architecture has many design points with the same peak throughput. The effective performance, however, varies with the number of tiles, DPEs, and lanes. To select an optimal NPU configuration for our in-depth study, we use our PSim to conduct design space exploration. Our PSim results show a large gap (up to 1.5$\times$) in cycle count between the best and the worst performing configurations at a given peak throughput.

Fig. 4 illustrates the pipeline traces in two different NPU configurations, which explains why some design points provide better performance than others. For example, with a 2T-960D-10L configuration, each tile needs to compute 960 columns, and it takes more than 96 cycles for the accumulators in MVU to output the final reduction value. Note that the first MFU computation can only start after that. In contrast, with a 4T-120D-40L configuration, each tile computes 480 columns, and the first MFU computation can start after 12 cycles only.

C. Optimizing NPU for Stratix 10 FPGAs

Based on the design space exploration, we chose the highest performing NPU configuration given the available FPGA resources for three different precisions: INT8, INT4 and FP32. We optimized our RTL to make the best use of FPGA resources and achieve a similar frequency to [4]. In this subsection, we will briefly describe some of our FPGA-specific NPU optimizations.

1) Arithmetic Units: For INT8 and INT4 precisions, we use sign-magnitude number representation to perform the multiplications. This enables us to pack four and six multiplications per DSP block for INT8 and INT4 precisions, respectively. The conversion of the input vector elements from two's complement to sign-magnitude is done on-chip before storing the values to VRFs while the model weight matrices are already stored in sign-magnitude representation in the MRFs. We then convert the results of the DPEs back to two's complement format for further processing. We also implement an FP32 NPU using DSP vector structures [13] to study how FPGAs perform even when using a GPU-friendly precision.

2) Data Movement: Each MVU tile in the NPU has a VRF that broadcasts the input vector elements to all its DPEs. In addition, although each DPE has its own MRF, a single address needs to be broadcasted to all MRFs due to their lock-step operation mode. This large fan-out is very unfriendly to the FPGA routing. To mitigate that, we implement a tree-shaped interconnect that reduces the fan-out of each node and introduces pipeline stages between the source and destination. We also implement a star-shaped interconnect for the paths from the central MVU logic to multiple MVU tiles and from the last MFU to multiple VRFs across the NPU, such that a distinct pipelined path is dedicated for each destination. We carefully wrote our RTL to enable the use of Stratix 10 HyperFlex registers to achieve higher operating frequency.

V. TensorRAM System-in-Package Chiplet

A. Chiplets for Stratix 10 System-in-Package

Intel Stratix 10 FPGAs leverage SiP solutions to integrate multiple ASIC components with an FPGA fabric in a single package. Stratix 10 SiP solutions utilize an embedded multi-die interconnect bridge (EMIB) and provide an efficient link between two chips, with $<$1 pJ/b energy consumption and $\sim$1 Tbps data streaming bandwidth [14]. Moreover, multiple EMIBs can be used to connect multiple heterogeneous components. Current products integrate an FPGA fabric with transceiver chips and HBM memory. These interconnected chips are often referred to as *chiplets* [14]. Such an architectural paradigm combines flexibility of FPGAs with efficiency of custom ASICs. Furthermore, it also offers other benefits that can be summarized as follows. First, it can re-use collaterals from the Stratix FPGA ecosystem (e.g. chips, packaging, boards, software) saving tremendous NRE cost and time-to-market. For instance, a new chiplet can be exposed to the

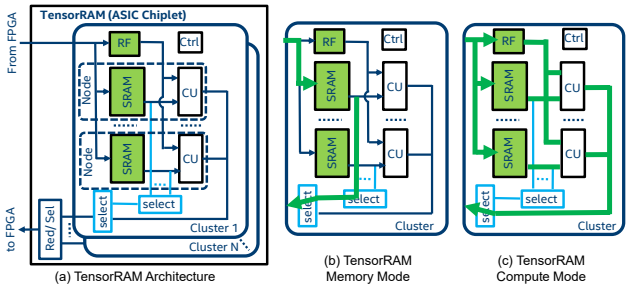

Fig. 5. The proposed TensorRAM chiplet architecture.

FPGA in the same way as any existing hard block (e.g. DSP block) through Intel Quartus® IP wizard. Second, SiP is scalable and customizable. A large Stratix 10 FPGA can be combined with up to 6 chiplets manufactured in different process technologies, allowing different levels of customizations.

B. TensorRAM Architecture

A TensorRAM chiplet is optimized to offer low-latency, energy-efficient, and high-bandwidth memory with near-memory compute units (CUs). Unlike the prior work proposing a compute-intensive DL chiplet called TensorTile [8], our TensorRAM chiplet complements the prior work by proposing a new data-oriented chiplet architecture that utilizes low-latency energy-efficient SRAMs. As shown in Fig. 5a, our TensorRAM chiplet contains low-latency register files (RFs) and SRAMs as well as a number of near-memory CUs matching the memory bandwidth. Each cluster consists of an RF and several nodes, each of which comprises an SRAM bank tightly coupled to a CU. For our study, CUs were chosen to be 64-wide INT8 DPEs that can be configured as 128-wide INT4, 256-wide ternary or 512-wide binary DPEs. The TensorRAM chiplet is designed with a form factor and interface that is compatible as a drop-in replacement for existing Stratix 10 SiP transceiver tiles. Thus, we target a TensorRAM configuration with 8 clusters and 64 nodes per cluster to meet our area and power constraints.

Our TensorRAM chiplet architecture can operate in a *memory mode*, acting like a typical RAM with deterministic read and write latency, or in a *compute mode* where computation is done near the SRAMs. An example operation of both modes is illustrated in Fig. 5b and 5c. In memory mode, the TensorRAM acts like a typical random access memory. Based on the read address that specifies which cluster and node to target, a particular SRAM bank is selected and data is read out from that bank using the bypass logic highlighted in light blue in Fig. 5. Similarly, to perform a write, it accepts write data and uses the write address to move the provided data to the target SRAM bank. In compute mode, each CU operates on a set of operands from the cluster-level RF and another set of operands from the coupled SRAM bank as shown in Fig. 5c.

The TensorRAM chiplet was designed to have a fairly simple interface and control mechanism. It contains a lightweight decoder which receives a command specifying either a memory read/write or a matrix-vector multiplication and translates it into a sequence of low level control signals to execute the specified operation. This is analogous to the concept of mOPs and μOPs discussed in Section III.

TABLE II
SUMMARY OF OUR EXPERIMENTAL SETUP

GPU	Nvidia Titan V GV100 (12nm)
FPGA-only	Intel Stratix 10 GX 2800 (14nm)
FPGA+ASIC	Intel Stratix 10 GX 1100 (14nm) + TensorRAM (10nm)
GPU Tools	CUDA Toolkit 10.0 + cuDNN 7.3
FPGA Tools	Intel Quartus Prime Pro 18.0
ASIC Tools	Synopsys Design Compiler 2018 + IC Compiler II 2017
Std. Cells	10nm FinFET CMOS characterized at 0.65V,100ºC [15]
Precisions	FP32 - INT8 - INT4
Benchmarks	DeepBench + Nvidia persistent RNNs [5]

C. Using TensorRAM with our NPU

Since the TensorRAM chiplet implements near-memory dot products, the NPU can offload its MVU functionality to the TensorRAM. The NPU sends/receives data to/from the TensorRAM-based MVU through the EMIB link which is exposed to the FPGA as a FIFO-based streaming interface, similar to what we already use to implement the original FPGA-based MVU. Hence, there is no need to re-architect the rest of the NPU to utilize TensorRAM as its MVU. As we offload matrix-vector multiplication on TensorRAM, we can achieve both better performance and area efficiency. First, since the ASIC chiplet operates at a faster clock frequency and has more memory and CUs, it can deliver higher performance for the MVU functionality. Second, since the MVU account for the majority of FPGA resources, offloading it to TensorRAM frees up resources to implement end-to-end AI applications on the FPGA as will be discussed in Section VII. Another alternative is to use a smaller FPGA combined with a chiplet to offer competitive performance with better efficiency and smaller form factor, which we evaluate in Section VI.

VI. Experimental Results

A. Methodology

For our study, we evaluate and compare the performance of a large GPU, our NPU on a large FPGA as well as on our FPGA-ASIC integrated solution using the TensorRAM SiP chiplet, all running persistent sequence models inference workloads. We choose our benchmark workloads from the DeepBench suite, used in [4], in addition to those used in the recent Nvidia persistent RNN paper [5] for a fair and direct comparison. Table II summarizes our experimental setup.

For GPU workloads, we use RNN, GRU and LSTM implementation from Nvidia cuDNN samples v7, which provides a variety of options that includes enabling persistency. RNN, GRU, and LSTM sizes are set according to the benchmark workloads in Table II. We sanity checked our persistent mode setup by replicating RNN experiments from the Nvidia paper [5]. The experiments in the Nvidia paper used a Tesla V100 GPU, which may have slightly different configurations for core and memory frequencies compared to Titan V, but we observe that our replicated results on Titan V are reasonably close to the ones reported in [5]. We were only able to collect FP32 results, since using INT8 precision resulted in a cuDNN internal error. We verified with the Nvidia authors from [5] that persistent INT8 is not supported by cuDNN to date.

To evaluate the performance of our FPGA-ASIC integrated solution, we implement all of the NPU blocks on the FPGA fabric except for the MVU block and its instruction decoder,

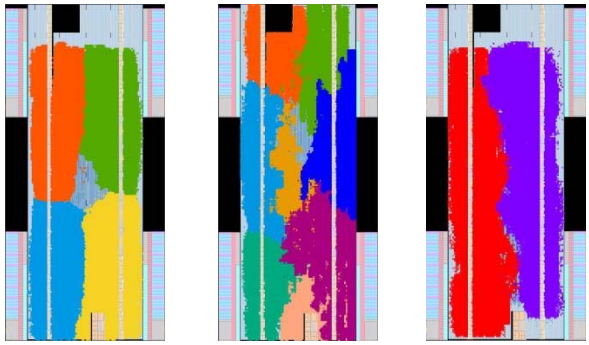

Fig. 6. NPU implementations on a Stratix 10 2800 FPGA.

TABLE III
IMPLEMENTATION RESULTS OF OUR NPU ON A STRATIX 10 2800 FPGA.

NPU Configuration	ALMs	M20Ks	DSPs	Freq. (MHz)	Peak TOPS
INT8 4T-120D-40L	567,982 (61%)	9,018 (77%)	4,880 (85%)	275	10.6
INT4 8T-120D-40L	786,295 (84%)	5,508 (47%)	5,600 (97%)	255	19.6
FP32 2T-64D-32L	286,024 (31%)	4,441 (38%)	4,768 (83%)	200	2.5

which are offloaded to the TensorRAM chiplet. In order to capture the routing effects on the FPGA side, we specify an empty logic-locked region partition for the MVU tile and decoder. We then place this logic-locked region close to the interface of the transceiver tile where the TensorRAM would be placed. This setup would create routing congestion that is similar to when routing signals from and to the chiplet.

B. FPGA-only Implementation Results

Fig. 6 shows our NPU implementations in chip planner for INT8, INT4, and FP32 precisions. The colors show different MVU tiles depending on the configuration we decided to implement in each of the three cases based on our design space exploration discussed in Section IV. Table III summarizes the configurations and FPGA implementation results for our different NPU versions. Although we achieve a similar frequency to that reported in the original BW paper [4] for the INT8 and INT4 variations, we believe that further low-level optimizations can be applied to our NPU implementation to achieve even higher operating frequencies. However, we leave that for future work since it has very limited impact on the conclusions we draw from our study in this paper.

C. TensorRAM ASIC Chiplet Results

Fig. 7 shows the layout of our TensorRAM ASIC chiplet detailed in Section V. Both SRAMs and RFs in TensorRAM used 10nm IPs [16]. The 31.8 mm^2 chiplet has a total RAM capacity of 64 MB consuming ∼80% of its area and delivers a peak performance of 64 INT8 TOPS at 1 GHz frequency. Since the CUs in TensorRAM can be configured to operate in different precisions, it has 128, 256 and 512 TOPS peak performance for INT4, ternary and binary precisions, respectively. Power consumption is estimated at maximum throughput for full chip utilization, with average switching activity in logic and memory cells. Synthesis and automated place & route were performed for the TensorRAM using libraries characterized at

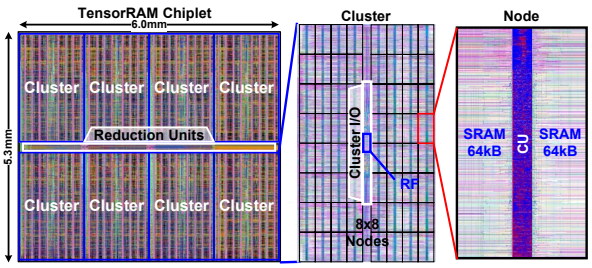

Fig. 7. Layout for the TensorRAM ASIC chiplet.

TABLE IV
COMPARISON BETWEEN TENSORRAM AND HBM2.

Parameter	TensorRAM	HBM2
Latency	$<$ 8 ns	16 ns - 32 ns *
Access Energy	$<$ 1 pJ/bit	2.24 pJ/bit - 3.45 pJ/bit *
Capacity	64 MB	4 GB - 8 GB (per stack)
Peak Bandwidth	256 GB/s	256 GB/s

* Numbers are reported for both row buffer hit and miss cases

0.65V, 100ºC [15]. Mixed-Vt libraries are used for optimum clock frequency and power trade-off. TensorRAM consumes 28.3 W resulting in energy efficiency of 2.3 TOPS/W for the INT8 and 18.1 TOPS/W for the binary modes.

To highlight the value of TensorRAM memory mode, we compare it with alternative SiP memory solutions. Stratix 10 MX devices include High Bandwidth Memory (HBM) connected via EMIB. Table IV summarizes pertinent memory parameters of both TensorRAM and HBM2 chiplets. Overall, TensorRAM provides over $2-4\times$ and $2-3\times$ better latency and energy efficiency than HBM2 respectively, at the cost of much smaller memory capacity. The read/write latency of HBM2 is approximately 16 ns in the case of row buffer hits [17], while that of TensorRAM is less than 8 ns. In terms of energy, HBM2 typically consumes a few pico joules to access a bit. For example, the HBM memory in GPUs consumes 2.24 pJ/bit to move data from the row buffer to the I/O pins, with an additional 1.21 pJ/bit for activation when the data is not in the row buffer [17]. In contrast, the energy to move data from an SRAM bank to the I/O pin in our TensorRAM, including wire interconnects and buffers in addition to SRAM access energy, is in the range of hundreds of femto joules. For bandwidth, both offer the same 256 GB/s bi-direction bandwidth limited by EMIB links. Thus, HBM2 is more suitable for workloads with large memory footprint, such as large graph analytics, while TensorRAM can be beneficial for latency-critical low-energy ones, such as network packet processing.

D. Comparative Analysis

Table V shows the latency comparison of our benchmarks on the GPU, FPGA, and FPGA+TensorRAM SiP along with the speedup relative to the GPU FP32 performance. The FPGA performance is consistently higher than that of the GPU. Even when using the GPU-friendly FP32 precision and operating at only 200 MHz, the FPGA achieves 2.7× lower latency than the GPU on average. This gap further increases to 8.6× and 11.2× when the FPGA uses INT8 and INT4 precisions at 250 MHz. These speedups are due to the low utilization of the GPU compared to that of the FPGA as shown in Fig. 8. The GPU is extremely underutilized, using only 6% of its peak TOPS on

TABLE V
RESULTS FOR THE GPU, FPGA AND FPGA+TENSORRAM SIP

Workload	Platform	Latency (ms)	Speedup
RNN h=1152 t=256	FP32 GPU	1	-
	FP32 FPGA	0.742	1.3
	INT8 FPGA	0.210	4.8
	INT8 TensorRAM	0.109	9.2
RNN h=1792 t=256	FP32 GPU	1.38	-
	FP32 FPGA	1.749	0.8
	INT8 FPGA	0.433	3.2
	INT8 TensorRAM	0.141	9.8
LSTM h=256 t=150	FP32 GPU	0.44	-
	FP32 FPGA	0.164	2.7
	INT8 FPGA	0.110	4.0
	INT8 TensorRAM	0.082	5.4
LSTM h=512 t=25	FP32 GPU	0.15	-
	FP32 FPGA	0.079	1.9
	INT8 FPGA	0.027	5.6
	INT8 TensorRAM	0.021	7.1
LSTM h=1024 t=25	FP32 GPU	0.44	-
	FP32 FPGA	0.254	1.7
	INT8 FPGA	0.064	6.9
	INT8 TensorRAM	0.036	12.2
LSTM h=1536 t=50	FP32 GPU	5.7	-
	FP32 FPGA	1.062	5.4
	INT8 FPGA	0.246	23.2
	INT8 TensorRAM	0.102	55.9
GRU h=512 t=1	FP32 GPU	0.085	-
	FP32 FPGA	0.003	28.3
	INT8 FPGA	0.00145	58.6
	INT8 TensorRAM	0.00098	86.7
GRU h=1024 t=1500	FP32 GPU	12.5	-
	FP32 FPGA	11.774	1.1
	INT8 FPGA	3.139	4.0
	INT8 TensorRAM	1.828	6.8
GRU h=1536 t=375	FP32 GPU	29.94	-
	FP32 FPGA	6.063	4.9
	INT8 FPGA	1.454	20.6
	INT8 TensorRAM	0.633	47.3

TABLE VI
PERFORMANCE OF FPGA SIP ON LARGER WORKLOADS

Workload	Latency (ms)
RNN h=4608 t=256	0.399
RNN h=5632 t=256	0.580
LSTM h=2048 t=25	0.066
GRU h=2048 t=375	0.810
GRU h=2560 t=375	0.621
GRU h=2816 t=750	2.156

average. Consequently, the FPGA also achieves higher energy efficiency compared to the GPU.

On the other hand, our FPGA-ASIC integrated solution using a small Stratix 10 device and the TensorRAM chiplet offers substantially better performance. The INT8 performance is on average 15.9× and 1.9× better than the GPU and FPGA-only solutions, respectively. Moreover, as shown in Table VI and Fig. 8, TensorRAM allows executing larger workloads at INT8 precision that do not fit into the on-chip memory of both the GPU and FPGA. In BW, executing such large workloads requires narrower block FP8 precision (INT3 with shared exponent) or scaling out to multiple FPGAs. Overall, the utilization of TensorRAM increases with larger problems in which more parallelism can be extracted to keep all CUs busy, achieving 50−80% utilization as shown in Fig. 8.

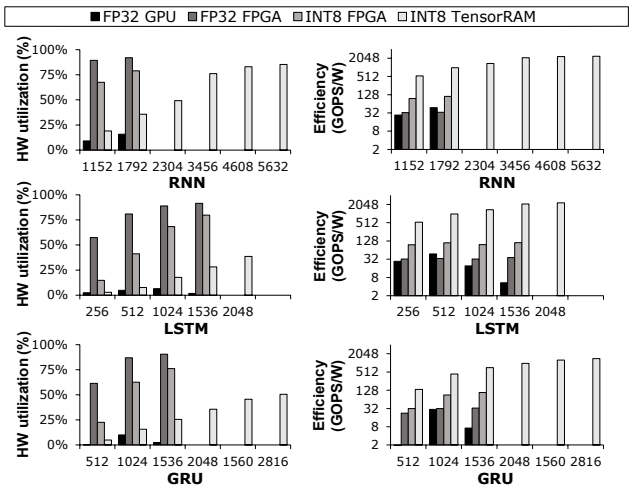

Fig. 8. Hardware utilization and energy efficiency of GPU, FPGA, and FPGA+TensorRAM SiP for different benchmark types and sizes.

TABLE VII
COMPARISON OF PLATFORM SPECIFICATIONS AND UTILIZATION. WORKLOADS ARE FROM SECTION VI, TR IS TENSORRAM, S10+ AND S10- ARE LARGE (GX2800) AND SMALL (GX1100) STRATIX 10.

Spec.	High-Performance			Energy-Efficient		
	TitanV GPU	S10+	S10+ w/ 6TR	T4 GPU	S10-	S10- w/ 1TR
Memory (MB)	34.5	28	412	10.9	10	74
Power (W)	250	120	290	75	24	54
INT8 Peak TOPS	60	28	393	130	10	64
Utilization	6%	57%	-	-	-	36%

VII. DISCUSSION

A. The Big Picture

A comparison between the specs of all three platforms for both high-performance as well as energy-efficient deployment targets is summarized in Table VII. For data-oriented workloads such as persistent DL processing, existing FPGAs are competitive to GPUs, offering similar on-chip memory capacity at lower power consumption. The proposed FPGA-ASIC integration with our TensorRAM chiplet offers not only higher performance but also better scalability. A large Stratix 10 FPGA can be integrated with up to 6 TensorRAM chiplets, offering 10× more on-chip RAM capacity than Titan V GPU, and 6.5× more peak INT8 TOPS, at only 1.16× the power consumption. For energy-efficient targets, a small Stratix 10 with one TensorRAM can offer 7× more on-chip RAM and 1.4× higher energy-efficiency compared the next-generation Nvidia T4 GPU, at 50% of its peak TOPS. However, our evaluation shows that the FPGA SiP can get reasonable 36% average utilization, while the Titan V GPU, with less INT8 peak TOPS than T4, is only getting 6% utilization.

B. Different Roles of the FPGA fabric and TensorRAM chiplet

By offloading the core NPU block to the TensorRAM chiplet, one might wonder whether the FPGA fabric would still be of any value. The key advantage offered by the FPGA fabric is flexibility through its fine-grained spatial programmability on the bit, cycle and dataflow levels. This motivates hardening the heavily-used operations in key application domains in

TABLE VIII
SUMMARY OF PRIOR WORK IN LITERATURE ON FPGA-BASED LSTM ACCELERATION

Work	[18]	[19]	[11]	[20]	[4]	This work	This work
Platform	SV GSMD5	Zynq Z7045	Kintex KU060	Zynq ZU7EV	S10 GX2800	S10 GX2800	S10 GX1100 with TR SiP
Model storage	Off-chip	On-chip	Off-chip	On-chip	On-chip	On-chip	On-chip
Model type	Dense	Dense	Sparse	Dense	Dense	Dense	Dense
Total params (M)	-	-	0.73	-	4.2	2.36	2.36
Precision (Bits)	16 fixed	5 fixed	12 fixed	8 / 1 fixed	BFP8	8 / 4 fixed	8 / 4 fixed
Frequency (MHz)	150	142	200	266	250	260 / 255	250+ 1000
Perf. (GOPS)	316	693	282	661 / 3725	22620	7980 / 14112	18481 / 18406
Efficiency (GOPS/W)	12.63	55.88	6.9	-	180	118 / 216	356 / 355

order to free up the precious FPGA fabric for other non-standard functionalities. Thus, we propose the TensorRAM chiplet which hardens the core operation of almost all DL algorithms (i.e. dot product) as we envision data-intensive DL algorithms to be a crucial portion of numerous end-to-end applications in wireless communications, finance, etc.

Recently, we are witnessing the emergence of more exotic AI models and applications such as neural machine translation (NMT) [21]. These applications include sequence models (i.e. RNNs or LSTMs) as a subroutine combined with other components such as word embedding, attention layers and different vector operations. In such case, our FPGA+TensorRAM proposal can be of substantial importance. Word embedding, which consists of dictionary lookups, can use the memory mode of our TensorRAM at much less energy and latency cost compared to DDR or HBM. Besides that, the TensorRAM compute mode with its high memory capacity and CU density can handle the matrix-vector operations in NMT's multi-layered LSTMs. Finally, the FPGA fabric can be used to implement the rapidly changing attention layers and vector operations. This is just an example for a recent AI application that can make use of the different capabilities of our FPGA-ASIC integrated solution, and we expect more to appear in the near future if the current trend continues.

VIII. RELATED WORK

Table VIII presents a brief summary of the prior work that focused on FPGA-based acceleration of LSTM workloads. For the commonly-used INT8 precision, we achieve a throughput of 7980 GOPS, which is 12× higher than the best prior INT8 FPGA work that we are aware of [20]. Also at INT4, we offer a throughput higher than most prior work, even when compared to the work that uses binary precision [20]. The only prior work that provides a higher throughput is [4] which uses block FP8 precision. For our FPGA-ASIC integrated solution, the TensorRAM increases performance significantly compared to prior work and also achieves the best energy efficiency.

Persistent DL processing has been proposed for both GPU and FPGA. Prior GPU studies evaluated dense [6] and sparse [5] persistent RNN, with recent Nvidia cuDNN supporting dense RNN, GRU, and LSTM. For FPGA, commercially deployed Microsoft Brainwave [4] relies on the persistent approach to fit the entire model into one or multiple ethernet-connected FPGAs. [7] is the earliest FPGA work that we are aware of that studied GRUs with all weights on-chip, but did not coin the term persistence. This paper complements these prior work by offering evaluation of the latest Stratix 10 FPGA, on known precisions, in comparison to the latest GPU also with cuDNN persistent kernels enabled. We also propose a novel TensorRAM chiplet for persistent sequence models.

FPGA EMIB-based SiP has been discussed in [14], but not in the context of DL. Stratix 10 SiP is commercially available integrating transceiver chiplets and HBM, but not DL-targeted chiplets. A recent paper [8] proposed a TensorTile chiplet for Stratix 10. However, it focuses only on compute-intensive CNNs, while the TensorRAM chiplet proposed in this work targets data-intensive and persistent sequence models such as RNNs, LSTMs, GRUs, and potentially more exotic workloads such as NMT and transformer networks [22]. To our knowledge, except for [8], this paper is the only other work on FPGA integration with ASIC chiplets in SiP, and the only one targeting data-intensive DL sequence models.

In general, several studies focused on compute in/near memory [23], [24]. This approach was adopted for DL acceleration outside the context of FPGAs. Neural Cache [25] proposes a new CPU cache for bit-serial DL processing in memory. For FPGAs, [26] proposed the 3D-stacking of a DRAM tier and two FPGA tiers with special compute blocks. It showed that applications like fast Fourier transforms and quick sort benefit from the additional memory close to compute units. However, this was not evaluated for modern DL applications.

IX. CONCLUSION

In this paper, we make a case for FPGA-ASIC integration in the context of real-time persistent data-intensive DL workloads such as RNNs, GRUs and LSTMs. As a benchmark for our study, we implement a Brainwave-like NPU along with a complete development flow for functional and cycle-accurate simulation as well as design space exploration. We also implement TensorRAM, a new 10nm ASIC chiplet for persistent sequence models to be integrated with a Stratix 10 FPGA as system-in-package. Our 32 mm^2 TensorRAM chiplet offers 64 MB memory capacity, 32 TB/s on-chiplet memory bandwidth with 64 TOPS for INT8 and 2×, 4× and 8× higher TOPS for INT4, ternary and binary precisions, respectively.

We present a detailed comparison between the performance of the latest and largest Nvidia GPU, a large Stratix 10 FPGA and a small Stratix 10 FPGA coupled with a TensorRAM chiplet across a wide variety of benchmarks from the DeepBench suite. Our study shows that current Stratix 10 outperforms the GPU running FP32 persistent kernels by factors of 2.7× and 8.6× when using FP32 and INT8, respectively. Our proposed FPGA-ASIC integrated solution increases this gap to 15.9× for INT8 and achieves higher energy efficiency while freeing up a large portion of the FPGA fabric for implementing end-to-end applications that include DL sequence models as a subroutine such as NMTs.

REFERENCES

[1] A. Boutros *et al.*, "You Cannot Improve What You Do Not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference," *TRETS*, vol. 11, no. 3, 2018.

[2] N. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017, pp. 1–12.

[3] M. Gao *et al.*, "Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory," *ACM OS Review*, vol. 51, no. 2, 2017.

[4] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *ISCA*, 2018.

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

[6] G. Diamos *et al.*, "Persistent RNNs: Stashing Recurrent Weights On-Chip," in *ICML*, 2016.

[7] E. Nurvitadhi *et al.*, "Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC," in *FPL*, 2016.

[8] 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 *FPL*, 2018.

[9] M. Abdelfattah *et al.*, "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration," in *FPL*, 2018.

[10] E. Nurvitadhi *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" in *FPGA*, 2017.

[11] S. Han *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *FPGA*, 2017.

[12] B. Pasca and M. Langhammer, "Activation Function Architectures for FPGAs," in *FPL*, 2018.

[13] M. Langhammer and B. Pasca, "Floating-Point DSP Block Architecture for FPGAs," in *FPGAs*, 2015.

[14] S. Shumarayev, "Heterogeneous Modular Platform," in *Hot Chips*, 2017.

[15] C. Auth *et al.*, "A 10nm High-Performance and Low-Power CMOS Technology Featuring 3rd Generation FinFET Transistors, Self-Aligned Quad Patterning, Contact Over Active Gate and Cobalt Local Interconnects," in *IEDM*, 2017.

[16] Z. Guo *et al.*, "A 23.6 Mb/mm2 SRAM in 10nm FinFET Technology with Pulsed PMOS TVC and Stepped-WL for Low-Voltage Applications," in *ISSCC*, 2018.

[17] M. O'Connor *et al.*, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *MICRO*, 2017.

[18] Y. Guan *et al.*, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *FCCM*, 2017.

[19] V. Rybalkin *et al.*, "Hardware Architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition," in *DATE*, 2017.

[20] V. Rybalkin *et al.*, "FINN-L: Library Extensions and Design Trade-off Analysis for Variable Precision LSTM Networks on FPGAs," in *FPL*, 2018.

[21] Y. Wu *et al.*, "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016.

[22] A. Vaswani *et al.*, "Attention is All You Need," in *NIPS*, 2017.

[23] D. Patterson *et al.*, "A Case for Intelligent RAM," *IEEE Micro*, vol. 17, 1997.

[24] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.

[25] C. Eckert *et al.*, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *ISCA*, 2018.

[26] P. Gadfort *et al.*, "A Power Efficient Reconfigurable System-in-Stack: 3D Integration of Accelerators, FPGAs, and DRAM," in *SOCC*, 2014.