A Configurable Cloud-Scale DNN Processor for Real-Time AI

Jeremy Fowers Kalin Ovtcharov Michael Papamichael Todd Massengill Ming Liu
Daniel Lo Shlomi Alkalay Michael Haselman Logan Adams Mahdi Ghandi
Stephen Heil Prerak Patel Adam Sapek Gabriel Weisz Lisa Woods
Sitaram Lanka Steven K. Reinhardt Adrian M. Caulfield Eric S. Chung Doug Burger

Microsoft

***Abstract*—Interactive AI-powered services require low-latency evaluation of deep neural network (DNN) models—aka "real-time AI". The growing demand for computationally expensive, state-of-the-art DNNs, coupled with diminishing performance gains of general-purpose architectures, has fueled an explosion of specialized Neural Processing Units (NPUs). NPUs for interactive services should satisfy two requirements: (1) execution of DNN models with low latency, high throughput, and high efficiency, and (2) flexibility to accommodate evolving state-of-the-art models (e.g., RNNs, CNNs, MLPs) without costly silicon updates.**

This paper describes the NPU architecture for Project Brainwave, a production-scale system for real-time AI. The Brainwave NPU achieves more than an order of magnitude improvement in latency and throughput over state-of-the-art GPUs on large RNNs at a batch size of 1. The NPU attains this performance using a single-threaded SIMD ISA paired with a distributed microarchitecture capable of dispatching over 7M operations from a single instruction. The spatially distributed microarchitecture, scaled up to 96,000 multiply-accumulate units, is supported by hierarchical instruction decoders and schedulers coupled with thousands of independently addressable high-bandwidth on-chip memories, and can transparently exploit many levels of fine-grain SIMD parallelism. When targeting an FPGA, microarchitectural parameters such as native datapaths and numerical precision can be "synthesis specialized" to models at compile time, enabling high FPGA performance competitive with hardened NPUs. When running on an Intel Stratix 10 280 FPGA, the Brainwave NPU achieves performance ranging from ten to over thirty-five teraflops, with no batching, on large, memory-intensive RNNs.

***Index Terms*—neural network hardware; accelerator architectures; field programmable gate arrays**

I. Introduction

Hardware acceleration of deep neural networks (DNNs) is becoming commonplace as the computational complexity of DNN models has grown. Compared to general-purpose CPUs, accelerators reduce both cost and latency for training and serving leading-edge models. Fortunately, the high level of parallelism available in DNN models makes them amenable to silicon acceleration. With evolving DNN-specific features, GPGPUs have been particularly successful at accelerating DNN workloads. In addition, a Cambrian explosion of new Neural Processing Unit (NPU) architectures is taking place, driven by academic researchers, startups, and large companies.

Training and inference (evaluating a trained model) have different requirements, however. Training is primarily a throughput-bound workload and insensitive to the latency of processing a single sample. Inference, on the other hand, can be much more latency sensitive. DNNs are increasingly used in live, interactive services, such as web search, advertising, interactive speech, and real-time video (e.g., for self-driving cars), where low latency is required to provide smooth user experiences, satisfy service-level agreements (SLAs), and/or meet safety requirements.

Highly parallel architectures with deep pipelines, such as GPGPUs, achieve high throughput on DNN models by batching evaluations, exploiting parallelism both within and across requests. This approach works well for offline training, where the training data set can be partitioned into "minibatches", increasing throughput without significantly impacting convergence. However, systems optimized for batch throughput typically can apply only a fraction of their resources to a single request. In an online inference setting, requests often arrive one at a time; a throughput architecture must either process these requests individually, leading to reduced throughput while still sustaining batch-equivalent latency, or incur increased latency by waiting for multiple request arrivals to form a batch.

We have developed a full-system architecture for DNN inference that uses a different approach [1], [2]. Rather than driving up throughput at the expense of latency by exploiting inter-request parallelism, the system reduces latency by extracting as much parallelism as possible from individual requests. We do not sacrifice throughput but achieve it as the direct result of low single-request latency. We use the term "real-time AI" to describe DNN inference with no batching. This system, called Project Brainwave (BW for short) achieves much lower latencies than equivalent technologies such as GPGPUs on a watt-for-watt basis, with competitive throughput.

This paper details the architecture and microarchitecture of the BW NPU, which is at the heart of the BW system. In its current form, the BW NPU is a DNN-optimized "soft processor" synthesized onto FPGAs. Despite the lower clock rate and higher area overheads that FPGAs incur, the BW NPU achieves record-setting performance for real-time AI, sustaining 35 Teraflops on large RNN benchmarks with no batching. However, only one of the techniques that the BW NPU uses to achieve low latency on individual DNN requests is tied to reconfigurable logic, and the rest could be applied to a "hard NPU" with a higher clock rate but reduced flexibility.

The key aspects of the BW system and NPU are:

- **Architecture:** The BW NPU implements a single-threaded SIMD ISA comprised of matrix-vector and vector-vector operations, contrasting with most current accelerators which are heavily multithreaded (e.g., GPUs) or fine-grained MIMD (e.g., Graphcore [3]). While using a single-threaded model for a massively parallel accelerator may seem counterintuitive, the BW NPU is able to extract sufficient SIMD and pipeline parallelism to provide high utilization from individual requests. The single-threaded model also reduces the burden on software; rather than relying on compilers or programmers to extract parallelism, the software that runs on the BW NPU is primarily a linearization of the operators in the accelerated subgraph.
- **Memory system:** To minimize latency, the BW system typically pins DNN model weights in distributed on-chip SRAM memories, which provide terabytes per second of bandwidth at low power (as in other recent NPUs [4]). Our current FPGAs also have local DRAM, which can be used for more computationally intensive (and thus less bandwidth-bound) models such as CNNs, or to run large models on a limited number of FPGAs.
- **Microarchitecture:** The BW NPU microarchitecture uses three techniques to extract parallelism. The first is hierarchical decode and dispatch (HDD), where compound SIMD operations are successively expanded into larger numbers of primitive operations and fanned out to the functional units. As an example, a single compound matrix-vector instruction will end up producing over 10,000 primitive operations. Second, the BW NPU employs vector-level parallelism (VLP), where the compound operations are broken into operations with a fixed native vector size (e.g. 100-400 operations per vector); it is these vector operations that form the primitives that are fanned out to the compute units (similar to scalar instructions in an ILP machine). Third, the BW toolchain maps these parallelized vector operations to a flat, one-dimensional network of vector functional units, and connects them in a dataflow manner so vectors can flow directly from one functional unit to another to minimize pipeline bubbles. Higher-level operations such as matrix-vector multiplications or convolutions are superimposed onto the flat vector unit space.
- **System:** The FPGA chips that host BW NPUs in the BW system are connected directly to the datacenter network, so that they can receive streams of DNN inference requests from remote servers with little overhead. Models with multiple components can be partitioned across multiple FPGAs. While the BW system currently runs multi-FPGA models in production, the focus of this paper is on single-node evaluation of popular models that fit entirely within individual FPGAs.
- **Synthesis Specialization:** Since the BW NPU provides a high-level single-threaded abstraction, the underlying microarchitecture can vary widely. This flexibility allows the BW NPU to be mapped to a variety of implementations, such as different generations of FPGAs or ASIC technologies. However, this flexibility also permits the microarchitecture to vary within a specific technology type and generation. The BW NPU accepts four synthesis-time parameters that can optimize the microarchitecture resources for a particular DNN model or class of models. These parameters are data type (precision), native vector size, number of data lanes, and size of the matrix-vector tile engine. This parameterization allows a leaner microarchitecture for each of a number of model classes, as opposed to a hardened implementation, which must implement a superset of support to be general across a range of model types.

Across a range of medium to large RNN benchmarks, running on a Stratix 10 FPGA, the BW NPU achieves throughputs ranging from 11 to 35.9 Teraflops, at latencies under 4 ms. These results use a low-precision floating-point format that provides equivalent model scoring accuracy. More importantly, these results are achieved without batching, so the system can serve requests individually in real time. The BW system shows that, for DNN inference, systems need not sacrifice low latency to achieve high throughput.

II. Background

The BW system integrates with an existing hyperscale cloud infrastructure that runs production services with real-time AI requirements [5]. This section gives background on the salient features of the target datacenter architecture and the main layers of the DNN serving stack.

A. A Hyperscale Datacenter Acceleration Architecture

Figure 1 illustrates the components of a hyperscale datacenter. The acceleration architecture we describe is in large-scale, world-wide production. Every standard dual-socket server hosts one or more PCIe-attached accelerator cards that contain FPGAs and/or ASICs. The accelerator cards have direct access to the datacenter network and are placed in-line between the server NIC and the top-of-rack (TOR) switches. Using an on-chip RDMA-like lossless protocol, the accelerators can communicate point-to-point directly at low latency to any of the hundreds of thousands of servers located in the same datacenter. The datacenter architecture shown in Figure 1 enables accelerators to be logically disaggregated and pooled into instances of hardware microservices with no software in the loop. Once initialized and registered with a distributed resource manager, a given hardware microservice is published to subscribing CPUs in the system and accessed directly through an IP address.

The system of Figure 1 fundamentally influences the way in which accelerators are managed and architected at cloud scale. The CPU and FPGA resources devoted to a particular acceleration scenario can be scaled independently, avoiding stranded capacity and improving overall datacenter utilization. Large,

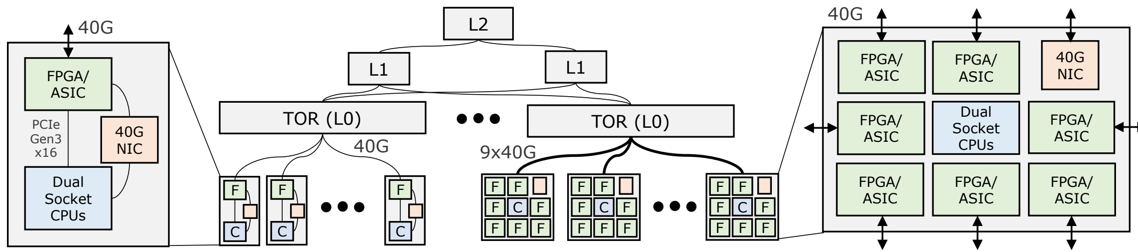


Fig. 1. BW system at cloud scale. From left to right, servers with bump-in-the-wire accelerators, accelerators connected directly to the hyperscale datacenter network, an accelerator appliance.

partitionable problems can be spatially distributed across multiple accelerators. For example, we have split bidirectional RNNs across two independent FPGAs, with the server invoking the forward and backward RNN FPGAs separately and concatenating their outputs.

B. DNN Acceleration Platform

The DNN acceleration platform runs on the scale infrastructure and consists of three components: (1) a toolflow that transforms pre-trained DNN model checkpoints into software and accelerator executables, (2) a federated runtime that orchestrates model execution between CPUs and distributed hardware microservices, and (3) the programmable BW NPU instantiated on FPGAs.

In the initial phases of the toolflow, a pre-trained DNN model is exported from a DNN framework (e.g., TensorFlow [6]) into BW's graph intermediate representation (GIR). The GIR undergoes a series of optimizations and transformations based on target constraints of the backend system, including the target number of accelerators and the available on-chip memory per accelerator. In latency-sensitive real-time scenarios, the toolflow can often partition large graphs that exceed the capacity of a single FPGA into sub-graphs whose parameters can be pinned individually into accelerators' on-chip memory. This partitioning avoids the memory capacity/bandwidth tradeoff that often thwarts the deployment of large RNNs and MLPs. Operations that are not supported by or cannot be profitably accelerated on the BW NPU are grouped into sub-graphs for execution on CPU cores.

Once the partitioning is complete, the FPGA and CPU subgraphs are compiled to BW NPU and CPU ISA binaries, respectively. (The BW NPU ISA is described in Section IV-C.) Once generated, the backend executables are packaged and deployed to a federated CPU runtime in the cloud that executes both the CPU sub-graphs and accelerator subgraphs initiated through calls to a remote hardware microservice.

III. Critical-Path Methodology for Latency-Aware Design

Before explaining the BW NPU architecture in detail, this section describes a critical-path methodology for rigorously guiding the design and evaluation of real-time NPUs optimized for the latency of single requests. While stand-alone metrics such as operations per second and energy efficiency are popular optimization targets, these metrics do not capture the effects of batching, which can artificially drive up utilization while increasing latency.

Model	Dimension	Ops	Cycles			Data
			UDM	SDM	BW NPU	
LSTM	2000x2000	64M	19	352	718	32MB
GRU	2800x2800	94M	31	520	662	47MB
CNN	In:28x28x128 K:128x3x3	231M	13	1204	1326	247KB
CNN	In:56x56x64 K:256x1x1	103M	13	549	646	200KB

TABLE I
Critical-path analysis of LSTM, GRU, and CNN.

To address this gap, we introduce additional latency-centric metrics based on critical-path analysis: (1) the number of cycles required to serve a model on an Unconstrained Dataflow Machine (UDM) with infinite resources, and (2) the cycles to serve on a Structurally-constrained Dataflow Machine (SDM) that shares the same number of functional units with a target accelerator implementation. When modeling the critical path, only functional unit latencies are counted in the UDM and SDM. These metrics enable robust NPU evaluation by characterizing the latency gap from idealized implementations. UDM reflects the lower bound latency capturing all available parallelism of a single DNN request; whereas SDM reflects the lowest possible latency under realistic resource constraints and assesses how well an implementation exploits available parallelism of a single DNN request with high microarchitectural efficiency.

LSTM Example. Figure 2 illustrates a critical-path analysis applied to the dataflow of a long short-term memory (LSTM) block [7], used commonly in state-of-the-art speech and text production DNN models. Table I compares the UDM and SDM latencies to the BW NPU on a single 2000-dimensional LSTM evaluation (see Section VII for more detail). The LSTM requires 64M operations per time step and can be executed in 19 cycles on an idealized UDM. The more realistic SDM constrained to 96,000 multiply-accumulators (MACs) as in the actual BW NPU, serves the model in 352 cycles. The 18X gap between the SDM and UDM suggests that further performance improvements can be gained with more resources. The actual BW NPU implementation serves the LSTM in 718 cycles—within 2X of the idealized SDM—indicating good microarchitectural efficiency. For reference,

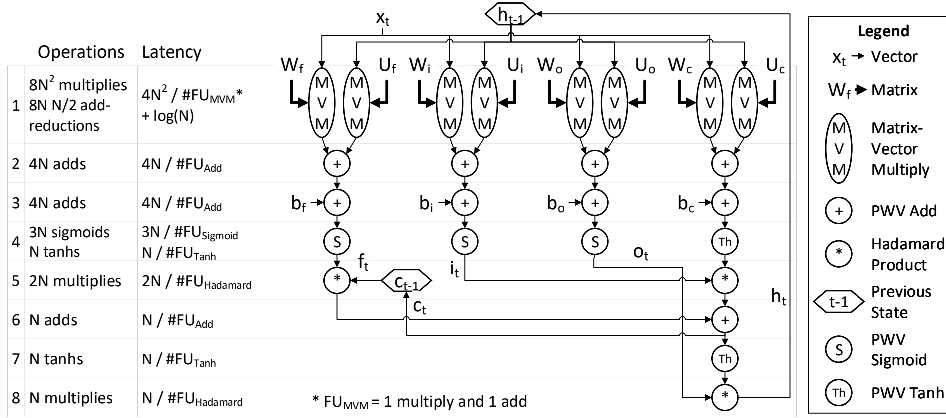


Fig. 2. LSTM critical-path analysis. Operation count and latency are shown as functions of LSTM dimension (N) and number of functional units (#FU).

Table I also compares LSTM to representative 2-D CNN layers from ResNet-50 [8]. The LSTMs overall are much more challenging to accelerate, exhibiting lower available parallelism and higher data requirements than 2-D CNNs.

Microarchitectural inefficiencies such as data and structural hazards, pipeline stalls, and limited memory bandwidth conspire to prevent NPU implementations from approaching ideal SDM latencies. The remainder of this paper shows that it is possible to extract most of the available parallelism from a single DNN request with high microarchitectural efficiency.

IV. Architecture

A. Overview

The goals of the BW NPU architecture are: (1) to provide a simple programming abstraction that can be targeted easily by programmers and compilers, (2) to encode sufficient information of potentially large DNN operations that an underlying microarchitecture can efficiently exploit available parallelism, and (3) to support enough flexibility to address a wide range of DNN models spanning RNNs, MLPs, and CNNs.

To meet these goals, the BW NPU adopts a single-threaded SIMD ISA made up of compound operations that operate on one- and two-dimensional fixed-size "native" vectors and matrices as first-class datatypes. While vector-based processing is a well-established paradigm, the BW NPU ISA introduces unique features tailored to the requirements of low-latency DNN serving. Sub-graphs of a large DNN model can be encoded through atomic instruction chains (without named storage between instructions) that efficiently capture explicit communication between graph edges, simplifying software development and reducing complexity in hardware (see Section V). The BW NPU architecture also exposes specialized instructions, datatypes, and memory abstractions that are optimized for low-latency DNN serving.

The BW NPU matrix/vector datapath is implemented as a coprocessor, using a conventional scalar core to issue BW NPU instructions to the datapath via an instruction queue. The scalar core provides the BW NPU's control flow, including dynamic input-dependent control flow, a critical requirement for certain models such as single-batch RNNs with variable-length timesteps.

B. Matrix-Vector Multiplication

In conventional basic linear algebra routines (BLAS) [9], there are three canonical levels at which BLAS linear algebra routines can be performed: (1) Level 1 (L1) consisting of vector-only operations, (2) Level 2 (L2) consisting of matrix-vector operations, and (3) Level 3 (L3) consisting of matrix-matrix operations. L1 can be used to implement L1-L3, and similarly L2 can be used to implement L2-L3.

The TPU1, a commercially deployed cloud NPU [10] most closely aligns to L3 using a systolic two-dimensional matrix-matrix multiplication array. L3 matrix-matrix multiplication is the most straightforward to scale in a hardware implementation due to high data re-use ($O(N)$) and reduced bandwidth requirements but requires dense layers in a DNN model to be batched for high efficiency. In other words, L3 matrix-matrix multiplication is a relatively poor fit for unbatched execution.

L2 matrix-vector multiplication, on the other hand, is highly desirable for single-request serving, especially in memory-intensive RNNs and MLPs that are dominated by large dense layers. As a result, the BW NPU architecture focuses on matrix-vector multiplication as its key operation. Many higher dimensional models (e.g., dense matrix-matrix multiply, 1D or 2D CNNs, etc.) can be efficiently linearized onto matrix-vector multiplication, whereas the opposite is more difficult. For the sake of optimized batch-1 serving and to support a variety of models, we specifically avoided the creation of higher-dimensional primitives such as explicit matrix-matrix multiplication and/or convolutional kernels. We have found that while optimizing lower-dimensional primitives in hardware is more difficult (but can be accomplished and scaled up to high degrees of parallelism as discussed in Section VII), the afforded flexibility has been invaluable.

C. BW NPU Instruction Set Architecture

The BW NPU instruction set has evolved over time to accommodate the requirements of production DNN models spanning LSTMs, GRUs, 1D (text) CNNs, 2D (image) CNNs,

Name	Description	IN	Operand 1	Operand 2	OUT
v_rd	Vector read	-	MemID	Memory index	V
v_wr	Vector write	V	MemID	Memory index	-
m_rd	Matrix read	-	MemID (NetQ or DRAM only)	Memory index	M
m_wr	Matrix write	M	MemID (MatrixRf or DRAM only)	Memory index	-
mv_mul	Matrix-vector multiply	V	MatrixRf index	-	V
vv_add	PWV addition	V	AddSubVrf index	-	V
vv_a_sub_b	PWV subtraction, IN is minuend	V	AddSubVrf index	-	V
vv_b_sub_a	PWV subtraction, IN is subtrahend	V	AddSubVrf index	-	V
vv_max	PWV max	V	AddSubVrf index	-	V
vv_mul	Hadamard product	V	MultiplyVrf index	-	V
v_relu	PWV ReLU	V	-	-	V
v_sigm	PWV sigmoid	V	-	-	V
v_tanh	PWV hyperbolic tangent	V	-	-	V
s_wr	Write scalar control register	-	Scalar reg index	Scalar value	-
end_chain	End instruction chain	-	-	-	-

PWV = point-wise vector operation. IN = implicit input (V: vector, M: matrix, -: none). OUT = implicit output.

TABLE II
THE SINGLE-THREADED BW NPU ISA EXPOSES A COMPACT AND SIMPLE ABSTRACTION FOR TARGETING DNN MODELS.

word/character embeddings, and dense MLPs. Table II gives a sampling of frequently used BW NPU instructions, discussed in greater detail below.

Data types, Storage, I/O. In the BW NPU, all instructions operate on N-length 1D vectors or $N \times N$ 2D matrices. Vectors and matrices are treated as separate data types and stored in independent register files. N is a fixed value in a given BW NPU implementation. The ideal vector size depends on the target set of models—a too-large vector requires inefficient padding, whereas a too-small vector increases control overhead. Section VI further discusses how synthesis specialization of native dimensions in the context of FPGA-based BW NPUs can be used to tailor a hardware instance to a model.

Vector and matrix read and write operations (v_rd, v_wr, m_rd, m_wr) use their first operand to select a memory target, which could be a specific register file, DRAM, or a network I/O queue. Their second operand provides a memory index, except in the case of network I/O. For other instructions, the memory structure accessed (if any) is implicitly identified by the opcode, as BW NPU memories are tightly coupled to specific function units. For these instructions, only an index operand is needed.

Matrix storage is specialized for constant values (model weights). Matrices can be read only from the network (for initialization) or from DRAM, and can be written only to the matrix register file (MRF) or to DRAM. The MRF is read only as an implicit operand of a matrix-vector multiply (mv_mul).

Instruction Chaining. A fundamental feature of the BW NPU ISA is explicit instruction chaining, in which sequences of dependent instructions pass values directly from one operation to the next. Explicit chaining allows the microarchitecture to exploit substantial pipeline parallelism without complex hardware dependency checking or multi-ported register files. As will be discussed further in Section V, this chain-enabled pipeline parallelism allows the microarchitecture to resolve critical serial dependences with low latency.

The IN and OUT columns in Table II show the implicit (chain) input and output operands for each instruction. Chains must begin with v_rd or m_rd, the only instructions which generate a chain output without a chain input. Vector chains may then include any number of operations that both consume and produce a vector, though the length and order of operations is constrained by the microarchitectural implementation. A vector chain terminates with a v_wr operation, which sinks the vector output of the prior instruction. As a special case of chain semantics, a vector chain can end with multiple v_wr operations, which multicasts the final vector value to multiple destinations.

Matrix chains always consist of exactly two instructions, an m_rd and an m_wr, and serve only to initialize matrices from the network and move matrices between the MRF and DRAM.

Mega-SIMD execution. The BW NPU enables operation on multiples of the native dimension using by setting scalar control registers using s_wr. For example, setting rows=4 and columns=5 causes subsequent mv_mul operations to treat 20 consecutive MRF entries as a tiled $4N \times 5N$ matrix, consuming 5 input vectors ($5N$ values) and producing 4 output vectors ($4N$ values). Other instructions in the chain are also scaled appropriately, e.g., the v_rd operation that feeds the mv_mul will read 5 contiguous VRF entries to provide sufficient input, and any v_wr at the end of the chain will write 4 contiguous VRF entries. This capability has been used with great extent to parameterize models, with the added benefit of reducing instruction bottlenecks. In the largest GRU evaluated in Section VII, a single instruction can be configured to dispatch over 7 million operations.

Despite its performance potential, the BW NPU ISA provides a succinct and relatively readable programming model. A fully parameterized and performance-tuned LSTM, summarized below, can be expressed in just under 100 lines of code:

```
void LSTM(int steps) {
  for (int t = 0; t < steps; t++) {
    v_rd(NetQ);
    v_wr( InitialVrf , ivrf_xt );
    // xWf = xt * Wf + bf
```

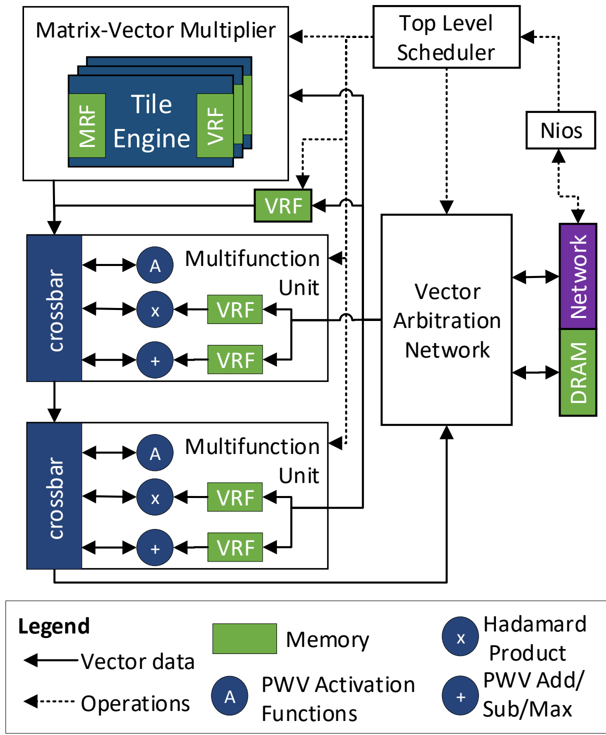


Fig. 3. Microarchitecture overview.

```
        v_rd( InitialVrf ,  ivrf_xt );
        mv_mul(mrf_Wf);
        vv_add(asvrf_bf);
        v_wr(AddSubVrf, asvrf_xWf);
        // xWi = xt * Wi + bi  ...
        // xWf = xt * Wo + bo ...
        // xWc = xt * Wc + bc ...
        // f gate  -> multiply by c_prev
        v_rd( InitialVrf ,  ivrf_h_prev );
        mv_mul(mrf_Uf);
        vv_add(asvrf_xWf);
        v_sigm();  //  ft
        vv_mul(mulvrf_c_prev);
        v_wr(AddSubVrf, asvrf_ft_mod);
        // i gate  ...
        // o gate  ...
        // c gate  -> store ct and c_prev
        v_rd( InitialVrf ,  ivrf_h_prev );
        mv_mul(mrf_Uc);
        vv_add(asvrf_xWc);
        v_tanh();
        vv_mul(mulvrf_it);
        vv_add(asvrf_ft_mod);  // ct
        v_wr(MultiplyVrf, mulvrf_c_prev);
        v_wr( InitialVrf ,  ivrf_ct );
        // produce ht,  store  and send to  network
        v_rd( InitialVrf ,  ivrf_ct );
        v_tanh();
        vv_mul(mulvrf_ot);  // ht
        v_wr( InitialVrf ,  ivrf_h_prev );
        v_wr(NetQ);
    }
}
```

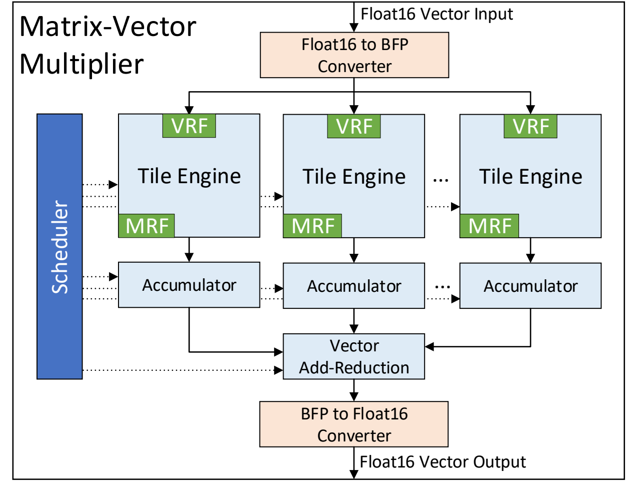


Fig. 4. Matrix-vector multiplier overview.

V. MICROARCHITECTURE

The goal of the BW NPU microarchitecture, as explained in the introduction and in Section III, is to maximally exploit the vector-level parallelism (VLP) of a single DNN request at high hardware efficiency. In practice, pipeline stalls caused by hazards, decoding inefficiencies, and inherent serial data dependences in models (e.g., between LSTM time steps) limit the exploitable VLP within a single request.

Figure 3 gives a high-level view of the BW NPU microarchitecture. The primary goal is to map and execute instruction chains (described in Section IV) to a continuous, uninterrupted stream of vector elements flowing through the function units. The function units form a linear pipeline, mirroring the instruction chain structure, with the matrix-vector multiplier (MVM) at the head. The vector arbitration network manages data movement among the memory components: pipeline register files (MRF and VRFs), DRAM, and network I/O queues. A top-level scheduler configures the control signals for the function units and vector arbitration network based on the BW ISA instruction chains it receives from the scalar control processor (a Nios soft processor in our current implementation).

The remainder of this section presents additional detail on three key aspects of the microarchitecture: the MVM, the vector multifunction units, and the hierarchical control structure that drives them.

A. Matrix-Vector Multiplier

The matrix-vector multiplier (MVM) is the primary workhorse of the BW NPU. The MVM is scaled across a network of dot product units consisting of up to 100,000 independent multipliers and accumulators. Unlike a matrix-matrix multiplier, the MVM is memory-bandwidth limited. To alleviate this bottleneck, each input to every single dot product unit requires a dedicated memory port to feed the units at maximum throughput.

A hierarchical view of the MVM is depicted in Figure 4. At the highest level, the MVM instantiates a series of matrix-vector tile engines, each of which implements a native-sized MVM. In turn, each tile engine is made up of a series of dot

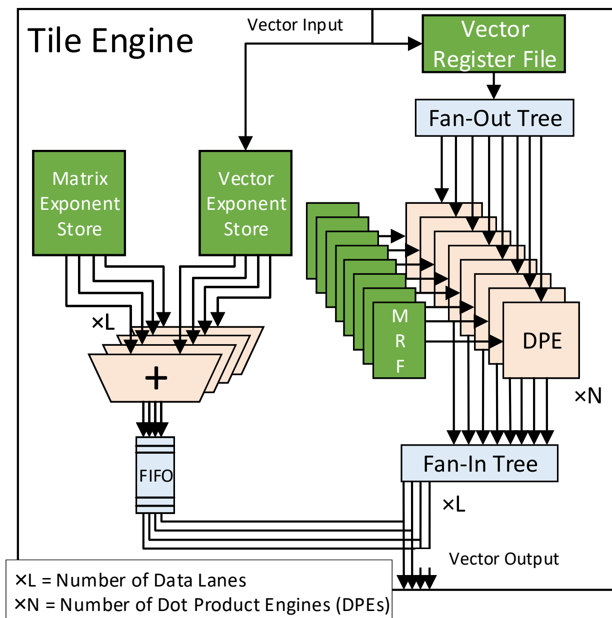


Fig. 5. Matrix-vector tile engine microarchitecture.

product engines, as shown in Figure 5. As the name suggests, each dot product engine is responsible for the dot product computation that corresponds to multiplying the input vector by one native row in the matrix tile. The dot product engines are composed of lanes of parallel multipliers that feed into an accumulation tree. These lanes provide parallelism within the columns of a row of a matrix tile. Combined, the MVM exploits four dimensions of parallelism: inter-MVM, MVM tiling, across the rows of a tile, and within the columns of the row. The total throughput of an MVM in FLOPs per cycle can be expressed as 2 $\times$ # tile engines $\times$ # DPEs $\times$ # lanes.

Sustaining L2 MVM Bandwidth. Each multiplier receives one vector element and one matrix element per cycle. The matrix elements are delivered by a dedicated port of the matrix register file (MRF) positioned adjacent to that multiplier. The MRF is banked by native tiles across the tile engines. Each bank is further sub-banked by rows, such that the first sub-bank in a tile engine (associated with the first dot-product engine) contains the first row of every matrix tile in the MRF bank. The elements of each row are interleaved in SRAM such that each SRAM read port can be directly connected to a multiplier.

This organization scales the number MRF read ports with local compute tiles. MRF entries can be written only from DRAM or the network input queue, so write bandwidth requirements are much lower.

Because MVM performance depends on MRF bandwidth, needed matrix operands must fit in the available on-chip SRAM to achieve high utilization. As discussed in Section II, the full-system design of the BW NPU architecture permits partitioning large multi-component RNN/MLP models across multiple accelerators when a single accelerator's on-chip memory is exhausted. CNNs are more compute intensive, and thus can overlap transfers of new operands from DRAM with computation on the current MRF contents.

B. Multifunction Units

The output from the MVM is routed through a series of vector multifunction units (MFUs). The MFUs support vector-vector operations such as multiplication and addition as well as unary vector activation functions like ReLU, sigmoid, and tanh. These are exposed in the ISA as the vv_* and v_* operations in Table II. Dedicated vector register files (VRFs) associated with the add/subtract and multiply function units provide the secondary operands needed for those operations.

Each MFU contains a set of vector function units (three in the current design) connected to the MFU's input and output ports by a non-blocking crossbar. The crossbar is configured according to the current instruction chain to route the MFU's input to its output via any sequence or sub-sequence of the internal function units (including a complete bypass). Once configured, a sequence of vectors can be pipelined through the MFU. Multiple MFUs can be chained to support longer sequences of vector operations. We have found that two MFUs are sufficient to support most instruction chains.

C. Hierarchical Decode and Dispatch (HDD)

The BW NPU uses a conventional scalar control processor (with its own scalar instruction set) to dynamically issue BW NPU instructions asynchronously to the top-level scheduler, as shown in Figure 3. In our current FPGA implementation, the control processor is realized with an off-the-shelf Nios II/f soft processor paired with custom C libraries for generating BW NPU instructions through software macros.

The top-level scheduler must decode each instruction chain into thousands of primitive operations to control the operation of many spatially distributed compute resources. The hierarchical decode and dispatch (HDD) logic shown in Figure 6 expands compound operations into distributed control signals that manage thousands of compute units and dozens of register files and switches. The top-level scheduler dispatches to 6 decoders and 4 second-level schedulers, which in turn dispatch to an additional 41 decoders. This scheme, combined with buffering at each stage, keeps the entire compute pipeline running with an average of one compound instruction dispatched from the Nios every four clock cycles.

Figure 6 illustrates how the largest functional unit (the MVM) is controlled from a single instruction. An expansion of decoding information occurs from top-to-bottom as the Nios processor streams T iterations of N static instructions into the top-level scheduler. Next, the top-level scheduler dispatches the MVM-specific portion of instructions to a second-level scheduler, which expands operations along the target matrix's R rows and C columns (using the row and column control registers described in Section IV-C). These MVM schedules are mapped to E matrix-vector tile engines, with operations dispatched to a set of E decoders each for the tile engines and their associated vector register files and accumulation units, along with a monolithic add-reduction unit. Finally, these decoders generate control signals that fan out into the data plane, with each tile engine dispatcher fanning out to hundreds of dot product engines.

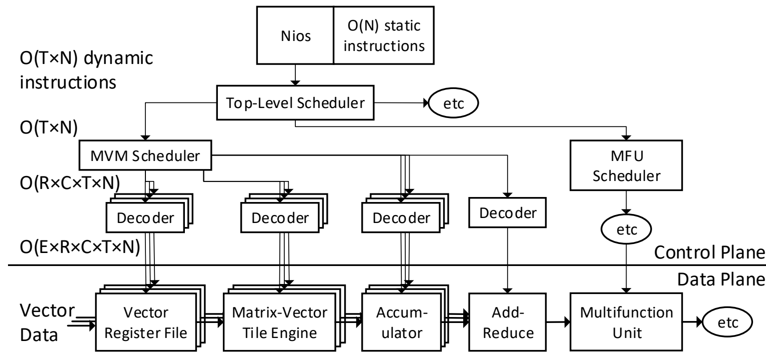


Fig. 6. Hierarchical decode and dispatch (HDD) into the matrix-vector multiplier.

VI. Synthesis Specialization

The BW microarchitecture can be viewed as a fully parameterizable processor family that can be customized to specific models for efficiency. While hardened BW processors can operate at high clock frequencies, their parameters must be selected at compile-time, whereas soft BW processors targeting FPGAs can be synthesis-specialized to particular models.

Datapath Specialization. The BW architecture exposes several major parameters that can be used for specializing a microarchitecture instance to specific models: (1) aligning the native vector dimension to parameters of the model tends to minimize padding and waste during model evaluation, (2) increasing lane widths can be used to drive up intra-row-level parallelism, (3) increasing matrix multiply tiles can exploit sub-matrix parallelism for large models. The results of Section VII show the implications for customizing parameters to models.

Narrow Precision Data Types. Extensive literature (e.g., [11]–[14]) has shown that deep neural networks are highly compressible in data types and sparsity. On FPGAs, we employ a narrow precision block floating point format [15] that shares 5-bit exponents across a group of numbers at the native vector level (e.g., a single 5-bit exponent per 128 independent signs and mantissas). The quantization noise introduced by BFP only affects dot products, since secondary operations (e.g., point-wise vector multiply or tanh) still execute as float16 on hardware. Using variations of BFP, we successfully trim mantissas to as low as 2 to 5 bits with negligible impact on accuracy (within 1-2% of baseline) using just a few epochs of fine-tuning on production state-of-the-art DNN models and large ImageNet models (e.g., ResNet-50). Using our variant of BFP, no hyperparameter tuning (e.g., altering layer count or dimensions) is required.

With shared exponents and narrow mantissas, the cost of floating point (traditionally the Achilles heel of FPGAs) drops considerably, since shared exponents eliminate expensive shifters per MAC, while narrow bitwidth multiplications map extremely efficiently onto lookup tables and DSPs. We employ a variety of strategies to exploit narrow precision to its full potential on FPGAs; for example, by packing 2 or 3 bit multiplications into DSP blocks combined with cell-optimized soft logic multipliers and adders, as many as 96,000 MACs can be deployed on a Stratix 10 280 FPGA.

VII. Evaluation

This section presents hardware implementation results across three generations of Intel FPGAs that demonstrate the scalability of the BW NPU architecture.

A. FPGA Implementation Results

We target three generations of Intel FPGAs, a mature — and relatively small by today's standards — Stratix V D5 device, and two contemporary FPGA devices, an Arria 10 1150 and Stratix 10 280[1]. All BW NPU instances have been validated against actual hardware and the reported resource usage and clock frequency results are based on final post-fit reports from Quartus 15.1.1 for Stratix V, Quartus Prime 17.0.0 for Arria 10, and Quartus Prime 17.1ir.2 for Stratix 10.

The high degree of synthesis-time parameterization described in Section VI allows us to tailor each BW NPU instance to satisfy the needs of the given model to be served and at the same time make the most efficient use of the underlying FPGA hardware. Table III shows HW implementation results for three BW configurations, BW_S5, BW_A10, and BW_S10, targeting three generations of Intel FPGA devices. Depending on workload and model requirements we can scale a BW NPU design across various dimensions until we exhaust the limiting resources on the given target FPGA. The remainder of this evaluation focuses on the BW_S10 instance.

B. RNN Performance Analysis

We physically measure the performance of the BW_S10 instance using DeepBench [16], a set of microbenchmarks containing representative layers from popular DNN models such as DeepSpeech [17]. We focus on GRU/LSTM inference tests at low batch sizes. We compare the latency and compute throughput trends between the BW NPU implemented on Stratix 10 280 and the DeepBench published results on a modern NVIDIA Titan Xp GPU.

Table IV shows a summary of the hardware used for the experiments. While both devices are made on similar process

[1]Stratix 10 results are measured on pre-production silicon.

BW NPU Instance	#MV Tiles	#Lanes	Native Dim.	MRF Size	#MFUs	Target Device	#ALMs (%)	#M20Ks (%)	DSPs (%)	Freq. (MHz)	Peak TFLOPS
BW_S5	6	10	100	306	2	Stratix V D5	149641 (87%)	1192 (59%)	1047 (66%)	200	2.4
BW_A10	8	16	128	512	2	Arria 10 1150	216602 (51%)	2171 (80%)	1518 (100%)	300	9.8
BW_S10	6	40	400	306	2	Stratix 10 280	845719 (91%)	8192 (69%)	5245 (91%)	250	48

TABLE III
HARDWARE IMPLEMENTATION RESULTS FOR DIFFERENT BW NPU CONFIGURATIONS AND FPGAS. NUMBERS IN PARENTHESES CORRESPOND TO THE FRACTION OF TOTAL DEVICE RESOURCES.

nodes, they are different in TDP and peak TFLOPS which is primarily due to the different numerical formats being used. We present both raw throughput and latency numbers and will draw conclusions based on percent hardware utilization, trends observed and comparison with an ideal dataflow machine.

	Titan Xp	BW_S10
Numerical Type	Float32	BFP (1s.5e.2m)
Peak TFLOPS	12.1	48.0
TDP (W)	250	125
Process	TSMC 16nm	Intel 14nm

TABLE IV
EXPERIMENT HARDWARE SPECIFICATIONS

1) No Batching: Batch size of 1 provides us with the lowest cloud service latency since requests are processed as soon as they arrive. It further simplifies software APIs and deployment complexity since a batching queues and runtime are not needed. Table V shows the raw effective TFLOPS and the execution latency of each benchmark. The BW NPU can run *all* DeepBench layers at under 4ms at batch 1, reaching up to **35.9 effective TFLOPS** for a large GRU over hundreds of timesteps. This represents an approximate two orders of magnitude advantage over the Titan Xp. This is in part attributed to the BW NPU's high peak TFLOPS at narrow precision, but more significantly, this is due to better hardware utilization. Figure 7 shows the utilization, which is the percentage of the peak TFLOPS reached for each layer. At batch size of 1, the BW NPU reaches 23% to 75% of peak FLOPS for medium to large LSTM/GRUs (>1500 dimension). This is a 4-23x improvement over Titan Xp's utilization. The BW NPU is able to fully expose on-chip RAM bandwidth, pipeline dependent RNN vectors and exploit all degrees of matrix-vector parallelism to keep its compute units busy. This translates into low latency response times at batch size of 1.

As the GRU/LSTM hidden dimension decreases, amount of parallelism and total operations decreases as well. Correspondingly, compute utilization drops for both Titan Xp and BW. In addition, for small LSTM/GRUs, BW's utilization drops due (1) the relative large native tile dimension, which can result in wasteful work, and (2) the deep pipelines that delays dependent data from being written back quickly. However, we emphasize that BW's reconfigurable architecture allows us to adjust for the different degrees of parallelism (e.g. shrink native dimension) according to the overall DNN dimensions, which can recover utilization and lower latency.

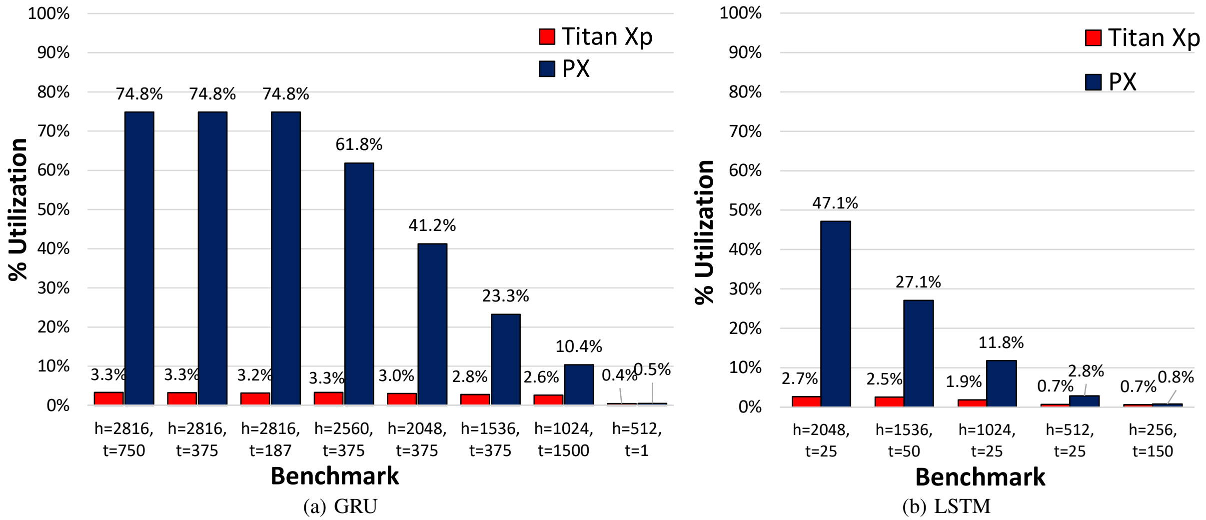
2) Critical Path Analysis: We use the critical path methodology from Section III to analyze the difference in latency between BW and a SDM with the same clock frequency and

Benchmark	Device	Latency (ms)	TFLOPS	% Utilization
GRU h=2816 t=750	SDM	1.581	-	-
	BW	1.987	35.92	74.8
	Titan Xp	178.60	0.40	3.3
GRU h=2560 t=375	SDM	0.661	-	-
	BW	0.993	29.69	61.8
	Titan Xp	74.62	0.40	3.3
GRU h=2048 t=375	SDM	0.438	-	-
	BW	0.954	19.79	41.2
	Titan Xp	51.59	0.37	3.0
GRU h=1536 t=375	SDM	0.266	-	-
	BW	0.951	11.17	23.3
	Titan Xp	31.73	0.33	2.8
GRU h=1024 t=1500	SDM	0.558	-	-
	BW	3.792	4.98	10.4
	Titan Xp	59.51	0.32	2.6
GRU h=512 t=1	SDM	0.00017	-	-
	BW	0.013	0.25	0.5
	Titan Xp	0.06	0.05	0.4
LSTM h=2048 t=25	SDM	0.037	-	-
	BW	0.074	22.62	47.1
	Titan Xp	5.27	0.32	2.7
LSTM h=1536 t=50	SDM	0.043	-	-
	BW	0.145	13.01	27.1
	Titan Xp	6.20	0.30	2.5
LSTM h=1024 t=25	SDM	0.011	-	-
	BW	0.074	5.68	11.8
	Titan Xp	1.87	0.22	1.9
LSTM h=512 t=25	SDM	0.0038	-	-
	BW	0.077	1.37	2.8
	Titan Xp	1.26	0.08	0.7
LSTM h=256 t=150	SDM	0.0126	-	-
	BW	0.425	0.37	0.8
	Titan Xp	1.99	0.08	0.7

TABLE V
DEEPBENCH RNN INFERENCE PERFORMANCE RESULTS

number of compute resources as BW_S10 (250 MHz, 96000 MVM MACs, and all other compute balanced). When we compare the SDM latencies in Table V to the BW_S10, we find that the BW_S10 is within a factor of 2.17X for the large GRUs and LSTMs (dimension > 2000). However, this factor falls off for the remaining models because the high-dimension MVMs and deep pipeline of the BW_S10 lead to essentially the same latency per time step in steady state for all evaluated models regardless of their size, between 252 and 296 microseconds. Future work will emphasize reducing the MVM granularity and increasing MFU resources to evaluate multiple spurs of the DNN graph in parallel as the SDM does.

3) Small Batches: Some cloud services will be able to tolerate a slightly longer response latency, in which case a small amount of batching can be employed. In Figure 8, we show utilization scaling with the number of batches. Though DeepBench caps inference batch size at 4, we also



(a) GRU

(b) LSTM

Fig. 7. Hardware utilization across DeepBench RNN inference experiments. h = hidden dimension, t = time steps

show batch size of 32 as a point of comparison. Since BW executes a single input at a time, with increased batch size, the utilization remains constant. We note that there is additional firmware optimizations to be made for batch size >= 2 by interleaving the computation for each RNN timestep among all input batches to further space out dependencies. This would be particularly effective at increasing utilization for small LSTM/GRU layers, which are not always able to fill the deep BW pipeline. This optimization is left for future work.

In contrast, GPU utilization increases proportionally as batch size increases since there is now more independent parallel work to fill the GPU SMs. However, at batch size of 4, the Titan Xp remains at under 13% utilization even for large RNNs. Increasing the batch size to 32 brings up GPU utilization, but in practice such large batch sizes cannot be used for DNN serving in the cloud without violating SLA.

4) Power Efficiency: We measured the peak chip power consumption of a Stratix 10 280 FPGA to be 125W by running a power virus design that used all of the on-chip IPs/resources. Though we did not measure power consumption for each DeepBench experiment, a conservative estimate based on the 125W peak power consumption figure, would put the power efficiency of BW at **287 GFLOPS/W** when running large models at high device utilization.

C. CNN Performance Analysis

The BW NPU architecture can also accelerate and serve large CNN models at low latency. In this section, we report on preliminary results of a BW NPU variant specialized for CNNs running a production image-based featurizer based on ResNet-50 [8]. The topology and computational requirements are nearly identical to the originally reported model except for the final dense layer, which is replaced by scenario-specific classifiers (e.g., decision tree in a Bing ranking pipeline) that run on CPU.

Table VI compares the latency and throughput to run the ResNet-50-based featurizer standalone on a BW NPU hosted

TABLE VI
THE BRAINWAVE NPU ON ARRIA 10 ACHIEVES COMPETITIVE THROUGHPUT AND LATENCY TO AN NVIDIA P40 GPU AT BATCH SIZE 1 ON A RESNET-50-BASED IMAGE FEATURIZER.

	Nvidia P40	BW_CNN_A10
Technology node	16nm TSMC	20nm TSMC
Framework	TF 1.5 + TensorRT 4	TF + BW
Precision	INT8	BFP (1s.5e.5m)
IPS (batch 1)	461	559
Latency (batch 1)	2.17 ms	1.8 ms

on an Arria 10 1150 (TSMC 20nm FPGA) against a high-end inference-optimized Nvidia P40 GPU (TSMC 16nm). Our measurements on real hardware include the latency to compute a single request as well as the transfer time over PCI express between host CPU and accelerator.

At batch size 1, the high-end P40 using INT8 precision achieves 461 inferences per second (IPS), while the BW NPU on Arria 10 achieves 559 IPS. On an unloaded system, the BW NPU serves a single instance of the model in 1.8 ms, while the P40 achieves 2.17 ms. The P40 achieves higher throughput than the Arria 10 at large batch sizes; for example, at a batch size of 16, the P40 attains 2,270 IPS; however, latency increases to 7 ms per batch, which does not include the time needed to form inputs from a batching queue. These results show that the BW NPU is an effective architecture for single batch, low latency serving—competitive with high-end, newer generation GPUs on compute-intensive CNNs and orders of magnitude faster on RNNs.

VIII. RELATED WORK

The unbridled successes of deep learning over the past several years (e.g., [8], [18]) have fueled an explosion of systems/AI research and the popularization of software frameworks for deep learning (e.g., TensorFlow [6], Caffe [19], etc). Correspondingly, many accelerators have been proposed, as we describe below. The combination of a single-threaded

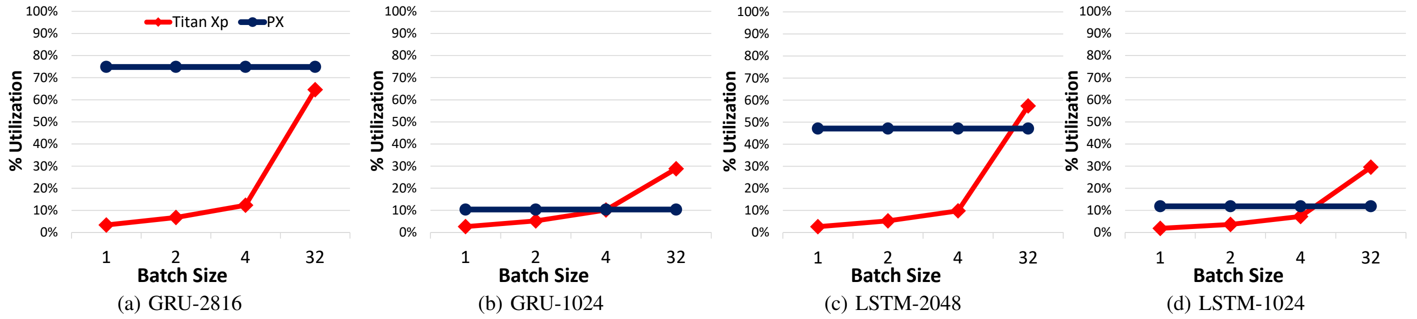


Fig. 8. Utilization scaling with increasing batch sizes.

programming model, a flat vector space suitable for CNNs and RNNs, model pinning in on-chip memories, and specialization to a range of vector lengths and data types at synthesis time differentiates the BW NPU from the large body of related work.

A plethora of ASIC-based DNN designs have focused on accelerating compute-intensive deep CNNs for computer vision (e.g., [20]–[32]). The 2-D CNN models exhibit high levels of parallelism and data re-use, making them easier targets for acceleration relative to memory-intensive RNNs.

The "DianNao" family of AI processors [4], [20], [33]–[35] span designs targeting small embedded clients to AI supercomputing, for both training and serving. The DaDianNao approach uses model pinning to minimize energy transfer and memory bottlenecks to feed spatially distributed compute units, a strategy similar to what we apply at datacenter scale. Eyeriss [36] explores a spatial dataflow engine for accelerating deep CNN models and achieves 10X more energy efficiency than GPUs. Other approaches also explore elision of unnecessary computation at the bit-level [37]–[39].

A significant percentage of cloud-scale workloads are driven by text-oriented scenarios requiring RNNs, MLPs, 1-D CNNs, and other more memory-intensive DNN algorithms, which many prior approaches do not consider. The EIE [12] and ESE [40] are two closely related works that optimize the serving of RNNs/MLPs through execution engines that process compressed models [41] directly. Deephi [42] offers production instances of CNN-optimized and RNN-optimized engines that compute on compressed models in FPGAs, but unlike the BW NPU they do not offer a converged engine that achieves high performance on uncompressed models.

The commercial viability of DNNs has spawned a wave of products from large companies and startups, including Google's TPU [10], [43], [44], Nvidia's SCNN [45], Wave Computing [46], Graphcore [3], Movidius [47], and many others (complete list in [48]). The TPU1 is the first reported large-scale deployed instance of a DNN accelerator for CNN and RNN inference. While it achieves high levels of efficiency for CNNs, it requires batching for high efficiency (at least 16) and performs with relatively low hardware utilization on RNNs (under 4%) even with high minibatch sizes [10].

Abstractions. An effort to generalize DNN accelerators such that they can support a wider range of DNN models has resulted in research into DNN-specific Instruction Set Architectures (ISAs). For example, Cambricon [49] is a load-store architecture that integrates data types such as scalars, vectors, matrices and control instructions to provide coverage for a wide range of DNNs to achieve higher code density than general-purpose ISAs as well as cover a wider range of models than traditional accelerators. The BW NPU uses a similar strategy, utilizing a specialized ISA optimized for low area footprint yet efficient execution of a wide range of DNN models.

Model Compression and Narrow Precision. Reduction in the numerical precision of activations and weights in neural networks is one strategy of achieving a boost in compute performance as well as power efficiency. Binarized Neural Networks and fixed-point numerical representation are popular approaches to achieving good efficiencies [11], [13], [14], [50]–[55]. However, this body of work is limited in that it only targets a single type of neural network, namely CNNs. To support a wider variety of neural networks, including RNNs, the BW NPU utilizes a novel approach to numerical quantization, one that allows it to scale the number of quantization bits given a target application, as well as utilize a shared vector exponent to enable a wider dynamic range, resulting in negligible or no losses in model accuracy.

Model compression and weight pruning are another approach to achieving good efficiencies. One such example is Deep Compression [41] but there are also others [56]–[61]. Although, these approaches can achieve great results for large, over-provisioned models, it is difficult to generalize to models of arbitrary size and complexity without significant drawbacks to model accuracy.

A key strategy used by the BW NPU to achieve high performance is through pinning neural networks—the idea that model weights can be pinned onto on-chip memory in order to achieve the necessary high memory read bandwidth for serving models in real time. Baidu's Persistent RNN [62] uses a similar approach targeting GPU devices for distributed training, although GPUs' on-chip memory is more limited in capacity and inflexible in precision. The BW NPU applies pinning to fully configurable FPGAs and uses a scale-out network of FPGAs to solve the case where the model weights

overflow the on-chip capacity.

There has been a large body of research in DNN accelerators targeting FPGAs, in particular for accelerating Convolutional Neural Networks (CNNs), such as [42], [63]–[94]. The BW NPU is one of the first to utilize the latest generation Intel Stratix 10 280 device for DNN acceleration achieving an order of magnitude higher performance than GPUs on RNNs [95].

Other research efforts also exist in analog and neuromorphic-type computing approaches such as [96]–[99]. However, these approaches typically do not provide state of the art accuracy compared to more artificial neural networks. For a more detailed survey on efficient DNN accelerators, refer to [48] and [100].

IX. Conclusions

The system that we described in this paper uses several techniques to achieve high throughput and low latency for real-time AI, with no minibatching. The system pins models in on-chip memories and extracts mega-SIMD parallelism from a single thread of control, with some of the compound instructions generating millions of independent operations. Hierarchical decode and dispatch breaks these operations into fleets of fixed-length vector operations that are then scheduled on a distributed substrate, operating in parallel and exploiting direct producer-consumer dataflow routing to reduce pipeline bubbles. Additionally, the datapath can be parameterized to provide a match to different models, whether it be data types, native vector size, number of lanes, or number of matrix-vector units. Taken together, these techniques allow higher utilization and lower latency on a collection of RNNs than a high-performance GPGPU built in an equivalent process technology. For the larger models, the latencies are 10-90X lower than the GPGPU, and effective utilization is higher than the GPU for all benchmarks until a batch size of 32 is applied. This system currently supports many models and is running in large-scale production (tens of thousands of nodes).

The dataflow analysis shows that there is still considerable parallelism in the larger models we measured to exploit. As the resources grow, so must the native vector length, so control overheads do not start to dominate. We do not yet know the limits of the single-threaded model to exploitable DLP times VLP, and whether this model will continue to scale to the increased area afforded by the few remaining silicon process nodes. However, the scaling from Stratix V, through Arria 10, and to Stratix 10 has been effective, with sustained utilizations. Another unknown in this space is the ideal clock rate; the FPGA clock rates are low compared to high frequency ASICs, allowing higher utilization of the architecture. As we push the frequency of the Stratix 10 implementation with production silicon, performance will grow but efficiencies will drop with increased pipeline bubbles. Similar to CPUs exploiting ILP, the NPU space must find the best balance of frequency and efficiency for exploiting vector-level processing, which is currently unknown.

References

[1] E. Chung *et al.*, "Accelerating persistent neural networks at datacenter scale," in *2017 IEEE Hot Chips 29 Symposium*, Aug 2017.

[2] ——, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," in *IEEE MICRO: Hot Chips*, April 2018.

[3] N. Toon and S. Knowles, "Graphcore," https://www.graphcore.ai, 2017.

[4] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *Proc. 47th Annu. Int. Symp. on Microarchitecture (MICRO)*, 2014, pp. 609–622.

[5] A. Putnam *et al.*, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *Proc. 41st Annu. Int. Symp. on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[6] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

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

[8] K. He *et al.*, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[9] C. L. Lawson *et al.*, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[10] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proc. 44th Annu. Int. Symp. on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[11] S. Gupta *et al.*, "Deep Learning with Limited Numerical Precision," in *Proc. 32nd Int. Conf. on Machine Learning - Volume 37*, 2015, pp. 1737–1746.

[12] S. Han *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[13] M. Courbariaux and Y. Bengio, "BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016.

[14] U. Köster *et al.*, "Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks," in *NIPS*, 2017.

[15] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, 1st ed. Englewood Cliffs, NJ: Prentice-Hall, 1963.

[16] S. Narang and G. Diamos, "Baidu DeepBench," https://github.com/baidu-research/DeepBench, 2017.

[17] A. Y. Hannun *et al.*, "Deep Speech: Scaling up end-to-end speech recognition," *CoRR*, vol. abs/1412.5567, 2014.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. 25th International Conference on Neural Information Processing Systems - Volume 1*, 2012, pp. 1097–1105.

[19] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proc. 22nd ACM International Conference on Multimedia*, 2014, pp. 675–678.

[20] Y. Chen *et al.*, "DianNao Family: Energy-efficient Hardware Accelerators for Machine Learning," *Commun. ACM*, vol. 59, no. 11, pp. 105–112, Oct. 2016.

[21] P. Whatmough, "DNN ENGINE: A 16nm sub-uj deep neural network inference accelerator for the embedded masses," in *2017 IEEE Hot Chips 29 Symposium*, Aug 2017.

[22] C. Farabet *et al.*, "Neuflow: A runtime-reconfigurable dataflow processor for vision," in *Proc. Embedded Computer Vision Workshop (ECVW'11)*, 2011, (invited paper).

[23] B. Moons and M. Verhelst, "A 0.3-2.6 TOPS/W Precision-Scalable Processor for Real-Time Large-Scale ConvNets," 2016.

[24] R. LiKamWa *et al.*, "RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, 2016, pp. 255–266.

[25] P. Chi *et al.*, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016, pp. 27–39.

[26] S. Chakradhar *et al.*, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," in *Proc. 37th Annu. Int. Symp. on Computer Architecture (ISCA)*, 2010, pp. 247–257.

[27] S. Venkataramani *et al.*, "ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in *Proc. 44th Annu. Int. Symp. on Computer Architecture (ISCA)*, 2017, pp. 13–26.

[28] S. Li *et al.*, "DRISA: A DRAM-based Reconfigurable In-Situ Accelerator," in *Proc. 50th Annu. Int. Symp. on Microarchitecture (MICRO)*, 2017, pp. 288–301.

[29] B. Reagen *et al.*, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016, pp. 267–278.

[30] M. Peemen *et al.*, "Memory-centric accelerator design for Convolutional Neural Networks," in *Proc. 31st IEEE Int. Conf. on Computer Design (ICCD)*, Oct 2013, pp. 13–19.

[31] S. W. Park *et al.*, "An Energy-Efficient and Scalable Deep Learning/Inference Processor With Tetra-Parallel MIMD Architecture for Big Data Applications," *IEEE Trans on Biomed Circuits Syst*, vol. 9, no. 6, pp. 838–848, Dec 2015.

[32] V. Gokhale *et al.*, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," in *2014 IEEE Conf. on Computer Vision and Pattern Recognition Workshops*, June 2014, pp. 696–701.

[33] T. Chen *et al.*, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *Proc. 19th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 269–284.

[34] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. 42nd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2015, pp. 92–104.

[35] D. Liu *et al.*, "PuDianNao: A Polyvalent Machine Learning Accelerator," in *Proc. 20th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2015, pp. 369–381.

[36] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016, pp. 367–379.

[37] P. Judd *et al.*, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. Int. Symp. on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[38] J. Albericio *et al.*, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016, pp. 1–13.

[39] ——, "Bit-pragmatic Deep Neural Network Computing," in *Proc. 50th Annu. Int. Symp. on Microarchitecture (MICRO)*, 2017, pp. 382–394.

[40] S. Han *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[41] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding," in *International Conference on Learning Representations*, 2016.

[42] K. Guo *et al.*, "From model to FPGA: Software-hardware co-design for efficient neural network acceleration," in *2016 IEEE Hot Chips 28 Symposium*, Aug 2016, pp. 1–27.

[43] C. Young, "Evaluation of the Tensor Processing Unit: A Deep Neural Network Accelerator for the Datacenter," in *2017 IEEE Hot Chips 29 Symposium*, Aug 2017.

[44] J. Dean, "Recent Advances in Artificial Intelligence via Machine Learning and the Implications for Computer System Design," in *2017 IEEE Hot Chips 29 Symposium*, Aug 2017.

[45] A. Parashar *et al.*, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 27–40, Jun. 2017.

[46] C. Nicol, "A dataflow processing chip for training deep neural networks," in *2017 IEEE Hot Chips 29 Symposium*, Aug 2017.

[47] D. Moloney, "Embedded deep neural networks: The cost of everything and the value of nothing," in *2016 IEEE Hot Chips 28 Symposium*, Aug 2016, pp. 1–20.

[48] V. Sze *et al.*, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[49] S. Liu *et al.*, "Cambricon: An Instruction Set Architecture for Neural Networks," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016, pp. 393–405.

[50] M. Rastegari *et al.*, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *In Proceedings of European Conference on Computer Vision*, 2016.

[51] B. Moons *et al.*, "Energy-efficient ConvNets through approximate computing," in *IEEE Winter Conf. on Appl. of Computer Vision (WACV)*, 2016, pp. 1–8.

[52] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 3123–3131.

[53] P. Judd *et al.*, "Reduced-precision strategies for bounded memory in deep neural nets," *CoRR*, vol. abs/1511.05236, 2015.

[54] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *CoRR*, vol. abs/1604.03168, 2016.

[55] P. Colangelo *et al.*, "Fine-grained acceleration of binary neural networks using Intel Xeon processor with integrated FPGA," in *Proc. 25th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 135–135.

[56] J. Cong and B. Xiao, *Minimizing Computation in Convolutional Neural Networks*. Cham: Springer International Publishing, 2014, pp. 281–290.

[57] J. Yu *et al.*, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," in *Proc. 44th Annu. Int. Symp. on Computer Architecture (ISCA)*, 2017, pp. 548–560.

[58] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6071–6079, 2017.

[59] Y. Kim *et al.*, "Compression of deep convolutional neural networks for fast and low power mobile applications," *CoRR*, vol. abs/1511.06530, 2015.

[60] F. N. Iandola *et al.*, "SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and <1MB model size," *CoRR*, vol. abs/1602.07360, 2016.

[61] S. Han *et al.*, "Learning Both Weights and Connections for Efficient Neural Networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 1135–1143.

[62] G. Diamos *et al.*, "Persistent RNNs: Stashing Recurrent Weights On-Chip," in *Proc. 33rd Int. Conf. on Machine Learning*, 2016, pp. 2024–2033.

[63] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN Accelerator Efficiency Through Resource Partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 535–547.

[64] J. Ouyang *et al.*, "SDA: Software-defined accelerator for large-scale dnn systems," in *2014 IEEE Hot Chips 26 Symposium*, Aug 2014, pp. 1–23.

[65] ——, "SDA: Software-defined accelerator for general-purpose big data analysis system," in *2016 IEEE Hot Chips 28 Symposium*, Aug 2016, pp. 1–23.

[66] J. Ouyang, "XPU: A programmable FPGA accelerator for diverse workloads," in *2017 IEEE Hot Chips 29 Symposium*, Aug 2017.

[67] D. Shin and H.-J. Yoo, "DNPU: An energy-efficient deep neural network processor with on-chip stereo matching," in *2017 IEEE Hot Chips 29 Symposium*, Aug 2017.

[68] A. Rahman, J. Lee, and K. Choi, "Efficient FPGA Acceleration of Convolutional Neural Networks Using Logical-3D Compute Array," in *Conf. on Design, Automation & Test in Europe (DATE)*, 2016, pp. 1393–1398.

[69] A. Podili, C. Zhang, and V. Prasanna, "Fast and efficient implementation of convolutional neural networks on FPGA," in *Proc. 28th IEEE Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, July 2017, pp. 11–18.

[70] S. Li *et al.*, "An FPGA design framework for CNN sparsification and acceleration," in *Proc. 25th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 28–28.

[71] L. Lu *et al.*, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. 25th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 101–108.

[72] Y. Shen, M. Ferdman, and P. Milder, "Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer," in *Proc. 25th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 93–100.

[73] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient FPGA implementation," in *Proc. 25th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 85–92.

[74] E. Kousanakis *et al.*, "An architecture for the acceleration of a hybrid leaky integrate and fire SNN on the convey HC-2ex FPGA-based processor," in *Proc. 25th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 56–63.

[75] S. Yin *et al.*, "Learning Convolutional Neural Networks for Data-Flow Graph Mapping on Spatial Programmable Architectures (Abstract Only)," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 295–295.

[76] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *Proc. 24th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 40–47.

[77] Y. Li *et al.*, "A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks (Abstract Only)," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 290–291.

[78] H. Nakahara *et al.*, "A Batch Normalization Free Binarized Convolutional Deep Neural Network on an FPGA (Abstract Only)," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 290–290.

[79] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[80] U. Aydonat *et al.*, "An OpenCL^{TM}Deep Learning Accelerator on Arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA'17. New York, NY, USA: ACM, 2017, pp. 55–64.

[81] Y. Ma *et al.*, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[82] C. Zhang and V. Prasanna, "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 35–44.

[83] J. Zhang and J. Li, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 25–34.

[84] R. Zhao *et al.*, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 15–24.

[85] E. Nurvitadhi *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 5–14.

[86] J. Qiu *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[87] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

[88] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[89] W. Qadeer *et al.*, "Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing," in *Proc. 40th Annu. Int. Symp. on Computer Architecture (ISCA)*, 2013, pp. 24–35.

[90] K. Ovtcharov *et al.*, "Toward accelerating deep learning at scale using specialized hardware in the datacenter," in *2015 IEEE Hot Chips 27 Symposium*, Aug 2015, pp. 1–38.

[91] C. Farabet *et al.*, "CNP: An FPGA-based processor for Convolutional Networks," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 32–37.

[92] C. Farabet, C. Poulet, and Y. LeCun, "An FPGA-based stream processor for embedded real-time vision with Convolutional Networks," in *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, Sept 2009, pp. 878–885.

[93] C. Ding *et al.*, "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices," in *Proc. 50th Annu. Int. Symp. on Microarchitecture (MICRO)*, 2017, pp. 395–408.

[94] M. Alwani *et al.*, "Fused-layer CNN accelerators," in *Proc. 49th Annu. Int. Symp. on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[95] D. Lewis *et al.*, "The Stratix^{TM}10 Highly Pipelined FPGA Architecture," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2016, pp. 159–168.

[96] D. Kim *et al.*, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016, pp. 380–392.

[97] A. Shafiee *et al.*, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *Proc. 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016, pp. 14–26.

[98] S. B. Eryilmaz *et al.*, "Neuromorphic architectures with electronic synapses," in *Proc. 17th Int. Symp. on Quality Electronic Design (ISQED)*, March 2016, pp. 118–123.

[99] S. K. Esser *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *CoRR*, vol. abs/1603.08270, 2016.

[100] A. Ling and J. Anderson, "The Role of FPGAs in Deep Learning," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2017, pp. 3–3.