VIP: A Versatile Inference Processor

Skand Hurkat[†]
Microsoft
Redmond, WA, USA
skand.hurkat@microsoft.com

José F. Martínez
Cornell University
Ithaca, NY, USA
martinez@cornell.edu

***Abstract*—We present Versatile Inference Processor (VIP), a highly programmable architecture for machine learning inference. VIP consists of 128 lightweight processing engines employing a vector processing paradigm, with a simple ISA and carefully chosen microarchitecture features. It is coupled with a modern, lightly customized, 3D-stacked memory system. Through detailed execution-driven simulations backed by RTL synthesis, we show that we can achieve online, real-time vision throughput (24 fps), at low power consumption, for both full-HD depth-from-stereo using belief propagation, and VGG-16 and VGG-19 deep neural networks (batch size of 1). Our RTL synthesis of a VIP processing engine in TSMC 28 nm technology, using a commercial standard-cell library supplied by ARM, results in 18 mm^2 of silicon area and 3.5 W to 4.8 W of power consumption for all 128 VIP processing engines combined.**

I. Introduction

Two classes of machine learning inference mechanisms that are of major interest are probabilistic graphical models (PGMs) and deep neural networks (DNNs). These algorithms are fundamentally different in the way they approach machine learning: PGMs are generative machine learning algorithms which model the underlying processes that produced the data, whereas most DNNs are discriminative algorithms that only label data. We can look into generative algorithms and reason about why they make decisions, whereas discriminative algorithms are largely black boxes. Inference on PGMs is used in computer vision for tasks such as image de-noising, depth-from-stereo, or detecting optical flow [15]; in biology, it is used to analyze DNA sequences [16] or to help pathologists diagnose lymph-node disease [20]. DNNs are used in a variety of other tasks, such as detecting objects in images and video [48], captioning images [54], and machine translation [59]. In both cases, these tasks often have low-power and real-time performance constraints, along with high computation and memory bandwidth requirements that have pushed their deployment from general purpose CPUs to GPUs, and lately to more and more specialized accelerators.

Recent accelerator proposals in the computer architecture community have in fact focused almost exclusively on DNNs. While there was interest in developing FPGA and ASIC accelerators for PGMs in the late 2000s and early 2010s [10, 12, 21], these accelerators are tailored to a narrow class of applications and data constraints (e.g., image size, number of categories being labeled, etc.). Accelerator proposals for PGMs include the Bayesian computing machine [30], which uses BRAMs within an FPGA for storage (which would require many FPGAs to scale up to larger problems), and Optical Gibbs' sampling [55], which involves inserting optical resonant units within GPUs for generating exponentially distributed random variables. While an interesting research proposition, the results presented in Optical Gibbs' sampling are based on projections of these optical resonant units being miniaturized to fit within a GPU—the prototype is a tabletop device using a laser and a resonant medium.

[†]Work developed while a PhD student at Cornell University.

Table I: Qualitative overview of various classes of existing architectures and systems, including CPU, GPU, TPU, FPGA, low-power ASICs, and this work (VIP) for inference on PGMs and CNNs. Lighter is better. An asterisk (*) indicates 24 frames per second or higher achieved at full-HD for stereo matching (PGM) or at VGG-16's standard size (CNN).

Platform	Power	Throughput		Programmability
		PGM	CNN	
CPU	Med/High	Low	Low	Very High
GPU	High	Med/High	High*	Very High
FPGA	Med	Med [21]	Med* [57]	Med
Tile-BP [10]	Very Low	Med/High	N/A	Very Low
Eyeriss [9]	Very Low	N/A	Low	Very Low
TPU [26]	Med	N/A	Very High*	Low
VIP	Low/Med	Very High*	Med*	High

Table I shows a qualitative overview of where a variety of relevant systems fall on a relative spectra of power, performance, and programmability. General-purpose architectures such as CPUs and GPUs are very programmable, but they require lots of power. Accelerators, on the other hand, provide good performance at low power, but do so with highly specialized designs that are not very programmable. FPGAs fall in the middle, but programming them requires writing a new RTL implementation—not an easy task.

In this work, we take a step back and ask ourselves: Surely highly specialized accelerators have their space in this landscape; but **is there a case to be made for a system that provides competitive performance at low power, for multiple classes of inference algorithms, while still remaining highly programmable?** Our results suggest that there is.

We present a new system, Versatile Inference Processor (VIP), designed for fast, efficient inference on both PGMs and DNNs. VIP has been designed to be highly programmable using the well-understood vector processing paradigm. We make economic but critical microarchitectural

choices, which we describe in this paper. We show the versatility and programmability of VIP by implementing inference workloads for PGMs, as well as two flavors of DNNs—convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs)—through software reprogramming alone.

The contributions of this work include:

1) We present an analysis of common kernels used for *a)* belief propagation on PGMs, and *b)* DNNs such as CNNs and MLPs. We describe how these kernels can be vectorized and discuss the similarities and differences in these workloads (Section II).
2) We present the VIP ISA, microarchitecture, and memory system, and we describe how the workloads described are implemented on VIP. We show that these very different workloads may be implemented on VIP through software reprogramming alone (Sections III and IV).
3) We evaluate VIP's performance using detailed microarchitecture and DRAM simulations. We find that VIP would exceed the performance of an Nvidia Titan X GPU for belief propagation on PGMs, and provide competitive performance against a hypothetical version of Eyeriss [9] scaled up to match VIP in area and technology (Sections V and VI).
4) We present an RTL synthesis of a VIP processing engine (PE) in TSMC 28 nm technology using a commercial standard-cell library supplied by ARM. We show that VIP's area and power requirements are modest; VIP's 128 PEs occupy 18 mm^2 and consume 3.5 W to 4.8 W while operating at 1.25 GHz (Section VII).

II. Workload Characterization

A. Probabilistic Graphical Models

Probabilistic graphical models (PGMs) represent probabilistic relationships between variables using graphs. Vertices represent random variables and the edges represent the joint or conditional probability distributions between these variables. PGMs called Markov random fields (MRFs) are often used in computer vision applications that are labeling tasks (assign labels to each pixel in the image), e.g., image segmentation (labels: objects), depth from stereo (labels: depth), or optical flow (labels: motion) [15]. The MRFs used in these applications are often 2D grid graphs (vertices corresponding to each pixel in the image) initialized with vertex (data) costs θ_v (vector) for each vertex v and edge (smoothness) costs $\theta_{v,w}$ (matrix) for each edge between vertices v and w. (Costs are related to negative logarithms of probability.) Belief propagation (BP) is an iterative algorithm for computation of probability in PGMs. Each vertex receives "messages" (probability information) from its neighbors. The vertex uses these messages to update its belief (probability distribution), and propagates this updated belief as new messages. Messages are computed as

$$\hat{\theta}_v(l_w) = \theta_v(l_w) + \sum_{x \in \mathcal{N}(v) \setminus w} m_{x \to v}(l_w) \tag{1a}$$

$$m_{v \to w}(l_v) = \min_{l_w} \left\{ \theta_{\{v,w\}}(l_v, l_w) + \hat{\theta}_v(l_w) \right\} \tag{1b}$$

where $m_{v \to w}$ is the message vector from vertex v to w, $\theta_{\{v,w\}}$ is the smoothness cost, θ_v is the data cost, and $\mathcal{N}(v)$ is the set of neighbors of v. (Equation (1b) is similar to a matrix-vector multiplication, but using a different set of operations.) When the messages converge, each vertex computes its most favorable label l_v.

$$l_v = \arg\min_{l_w} \left\{ \theta_v(l_w) + \sum_{x \in \mathcal{N}(v)} m_{x \to v}(l_w) \right\} \tag{2}$$

We use the accelerated BP algorithm (BP-M) proposed by Tappen and Freeman [50], as it converges fairly quickly while providing ample opportunities for parallelism. BP-M on a $I_x \times I_y$ image, using L labels requires $(4+1) \times L \times I_x \times I_y$ values be stored. A BP-M iteration requires $4 I_x I_y$ message updates, and each video frame requires n iterations. Each message update requires $3L + 2L^2$ operations and $4L$ data to be read or written. Depth from stereo for full-HD (1920×1080) video with sixteen labels at 24 fps (with 8 iterations per frame) will therefore require 316 MiB of data storage, 190 GiB s^{-1} memory bandwidth (with 16 bit data types) and 892 GOp/s of computational throughput.

B. Convolutional Neural Networks

The primary operation in convolutional neural networks (CNNs) is the convolution operation with a bias.

$$O(x,y,z) = f\left(b(z) + \sum_i \sum_j \sum_k I(x-i, y-j, k) h^z(i,j,k) \right) \tag{3}$$

where O is output feature map, I is the input feature map, h^z is the z^{th} convolution filter, and b is the bias. The convolution operation is highly parallel in the x, y, and filter (z) dimensions.

VGG-16 and VGG-19 networks [48] take 224×224 sized images as input and classify them into one of 1,000 categories using 13–16 convolution layers and three fully-connected layers. (The fully connected layers are discussed in Section II-C.) The thirteen convolution layers in VGG-16 require 15.3 billion multiply-accumulate (MAC) operations. At 24 fps, this translates to 734 GOp/s (1 MAC = 2 Op).

The max pooling operation in CNNs collects values from a neighborhood and combines these values through the maximum operator. This reduces the size of feature maps. The final operation in CNNs (which also appears in MLPs) is the activation operation. VGG-16 uses rectified linear unit (ReLU) function: $f(x) = \max\{x, 0\}$.

C. Multi-layer Perceptrons

Multi-layer perceptrons (MLPs), or fully-connected layers, consist of multiple layers of neurons where every neuron produces an output as a weighted sum of inputs. A layer in an MLP can be summarized as

$$O(i) = f\left(b(i) + \sum_j W(i,j) I(j) \right) \tag{4}$$

where O is the output vector, W is the weight matrix, I is the input vector, and b is the bias vector. An activation

function is applied to the outputs of each layer, similar to CNNs. VGG-16 [48] uses ReLU activation function for its fully connected layers.

Equation (4) involves two operations, a matrix-vector multiplication between the weights and inputs, and a vector-vector addition to add biases. As MLPs consist of matrix-vector multiplications, they are dominated by memory bandwidth requirements—e.g., the first fully connected layers in VGG-16 and VGG-19 networks take in 25,088 inputs and produces 4,096 outputs. With 16 bit data types, these require 196 MiB of data and 100 million MACs.

D. Why PGMs are Different than DNNs

There are some fundamental differences between computation on PGMs and on deep neural networks (DNNs).

First, PGMs operate on graphs and parallelism exists across different vertices in the graph. On the other hand, layers in CNNs and MLPs may be structured as monolithic matrix-vector or matrix-matrix multiplications (in the case of batched execution, a technique that improves data reuse). As a result, PGMs require a system that can exploit both fine-grained (computing each message using short vectors) and coarse-grained (across graph vertices using multiple processing engines) parallelism. The matrices in CNNs and MLPs are much larger, and may work with long vectors as well as tile matrices across multiple processing engines.

Second, most ISAs today support the MAC primitive as it is commonly used in matrix-matrix multiplication. The min-sum BP message update described by Equation (1b) uses a different composition of operations—addition of two vectors followed by a minimum reduction of the result. None of the accelerators for DNNs—e.g., Cambricon [33], Eyeriss [9], or Google's tensor processing unit (TPU) [26]—support this composition of operations.

In fact, both Cambricon and Google's TPU have a very limited datapath for true vector computations that are not matrix-matrix multiplications, as adding vectors occurs relatively rarely in CNNs, only in the pooling layers. Specifically, Cambricon has only 32 ALUs for vector operations, while it has 1024 MAC units for multiplying matrices. Cambricon will therefore require over 0.13 s just to compute Equation (1a) for one frame of a full-HD image, severely limiting its throughput (to less than 8 fps) on vector operations alone. Additionally Cambricon's microarchitecture is such that the matrices and vectors must already be in the scratchpads of the matrix unit, unlike PGMs which not only use a different set of operations, but also have tight dependencies between matrix and vector operations.

Lastly, many efficient BP computations on PGMs do not update vertices until all other vertices on the graph have been updated. Not only do PGMs have high memory bandwidth requirements, this also means that traditional techniques to exploit locality, such as caching, may not work well with PGMs due to long reuse distances.

E. Vectorizing PGMs, CNNs, and MLPs

Ni and Jain [38] define four types of vector operations—$f_1 : V \to V$, $f_2 : V \to S$, $f_3 : V \times V \to V$, and $f_4 : V \times S \to V$, where V and S denote sets of vector and scalar operands respectively. We observe, as discussed in subsequent paragraphs, that the operations involved in the workloads under consideration involve compositions of these categories of operations. We introduce shorthand notation for these compositions. $f_5 : V \times V \to V \to S$ is a composition of an f_3 operation followed by an f_2 operation. If we loop over an f_5 operation while keeping one of the vector operands constant, we create the category $f_6 : M \times V \to V$, where M represents the set of matrix operands. A vector dot-product is an example of an f_5 category operation, while a matrix-vector product is an example of the f_6 category. These compositions provide an efficient way to describe the operations in PGMs, CNNs, and MLPs, and are therefore an excellent basis for an ISA.

Operations in both PGMs and DNNs involve vector reduction and matrix-vector operations from categories f_2, f_5, and f_6. For example, Equation (1a) is an f_3 operation while Equation (1b) is an f_6 operation (although it uses a composition of operators distinct from multiply-add used in matrix-vector multiplication).[1] In a similar fashion, we can rewrite the convolution operation in CNNs—Equation (3)—as

$$R(x,i,y,j,z) = \sum_k I(x-i,y-j,k)h^z(i,j,k) \tag{5a}$$

$$Q(x,i,y,z) = \sum_j R(x,i,y,j,z) \tag{5b}$$

$$P(x,y,z) = \sum_i Q(x,i,y,z) \tag{5c}$$

$$O(x,y,z) = f\left(b(z) + P(x,y,z)\right) \tag{5d}$$

Again, we observe that Equation (5a) is an f_5 operation, Equations (5b) and (5c) are f_2 operations, and Equation (5d) consists of f_3 operations. Section II-C discussed how Equation (4) used in MLPs is an instance of an f_6 operation (multiply weights) and an f_3 operation (add bias).

Traditional vector processors are efficient at operations in categories f_1, f_3, and f_4, but not at reduction operations which are at the core of operations in categories f_2, f_5, and f_6. VIP attempts to fix this, as discussed in Section III-B. The other link to accelerating these applications: **improving the effective use of memory bandwidth**; we discuss in Section III-C how VIP employs a modern 3D stacked memory system with some modest customization to accomplish this.

III. System Description

VIP consists of multiple (128) processing engines (PEs) operating at 1.25 GHz located at the logic layer of a 3D

[1] Belief propagation on images with 16 labels results in vector and matrix dimensions of 16 and 16×16, respectively. These dimensions are much smaller than the ones encountered in neural networks or other applications involving matrix-matrix multiplication.

Table II: A summary of the VIP instruction set

Vector Instructions			
Configuration	`set.{vl,mr},{v.drain}`		
Matrix-vector	`m.v.{mul,add,sub,min,max.nop}.{add,min,max}`		
Vector-vector	`v.v.{mul,add,sub,min,max}`		
Vector-scalar	`v.s.{mul,add,sub,min,max}`		
Scalar Instructions		**Load-store Instructions**	
Reg-reg / reg-imm	`{add,sub,sll,srl,sra, and,or,xor}`	SRAM	`{ld,st}.sram`
		Reg	`{ld,st}.reg`
Move	`{mov,mov.imm}`	Sync	`memfence`
Control	`{blt,bge,beq,bne,jmp}`		

stacked memory system similar to Micron's Hybrid Memory Cube (HMC) [22]. Each VIP PE employs a vector processing paradigm. VIP provides a peak throughput ranging from 320 GOp/s for 64 bit data to 2,560 GOp/s for 8 bit data.[2] The HMC provides a peak memory bandwidth of 320 GB s^{-1}, which is sufficient for the high bandwidth requirements of probabilistic graphical models (PGMs) discussed in Section II-A. The HMC has 32 independent channels or vaults, four VIP PEs are placed in each vault, and vaults are connected using a 2D torus network. Figure 1 shows an overview of VIP's organization.

A. VIP ISA and Overview

Section II-E described how matrix-vector operations from the f_6 category are common in belief propagation (BP) on PGMs, and deep neural networks (DNNs) including convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs). Implementing these operations on traditional vector-SIMD machines, however, is tricky. There are two approaches to performing vector reductions: *a*) perform independent vector reductions in each vector element, or *b*) perform reductions in a divide-and-conquer manner, dividing the input vector in half every loop iteration and performing element-by-element operations using the two halves as inputs until only a single vector element remains. Matrix multiplication typically involves the first approach [23]. The matrices to be multiplied are tiled, with the tile size determined by the length and number of vector registers. Most applications involve large matrix multiplications, so the vector registers can be fairly large. As discussed in Section II-E, BP on PGMs typically works with short vectors, 32 B in the case of depth-from-stereo with 16 labels using a 16 bit data type. The IBM Active Memory Cube [37], as a representative vector processor, provides sixteen 256 B vector registers. Seven-eights of a vector register will be unoccupied if used to store just one vector in such a configuration, wasting space. In order to store vectors efficiently, multiple vectors must be packed into a single register, and unpacked as they are operated upon, adding overheads. While the ARM scalable vector extensions (SVE) ISA [4] does provide support for vector reduction instructions, it too is limited by the number of vector register names. Section VI-B discusses the extent of these overheads. GPUs, on the other hand, have their own inefficiencies. Each thread must perform redundant work such as computing addresses and executing loops. Additionally, GPUs rely on multi-threaded execution to hide instruction and memory latency; we will show in Section V-B that a large GPU such as the Titan X is limited in performance by lack of sufficient threads in BP-M.

VIP addresses the problem of many short vectors by changing its paradigm from vector-register (values must be in vector registers before they can be operated upon) to vector memory-memory (the processor can operate on values in memory). Modern DRAM memory, however, has unpredictable latency depending on the state of the DRAM. VIP provides a 4 KiB SRAM scratchpad to hold values being operated upon. The programmer must explicitly transfer values from DRAM to scratchpad before operating on these values. In this aspect, VIP lies between a vector-register and a true vector memory-memory paradigm. The scratchpad has multiple read and write ports just like a vector register file, the difference being that it can be accessed at any arbitrary location (memory address) instead of a fixed set of locations (register names). The programmer may load in as many vectors or matrices in the available space as they want, without losing efficiency due to alignment.

The applications considered use two nested loops to implement f_6 category operations, as discussed in Section II-E. While the outer loop may be executed in software, overheads such as bumping source and destination pointers, loop induction variables, and executing branches quickly become overheads as they cannot be hidden behind the execution of relatively short vectors. VIP offloads this work to hardware, allowing the system to perform f_6 category matrix-vector multiplication like operations. VIP allows the programmer to choose the composition of horizontal and vertical vector operators, providing them with the flexibility to work with different applications such as min-sum BP and sum-product matrix multiplication.

VIP provides a set of scalar instructions for executing loops, computing data addresses, etc. It also provides a set of instructions for moving data between DRAM and either scalar registers or the scratchpad. It does not provide a method for moving data between scalar registers and the scratchpad as the scalar unit is meant to operate on a different set of data than the vector unit. Both the scalar and load-store units are designed to operate in the shadow of the vector unit; a programmer can start a long-latency vector operation and perform additional tasks such as prefetching the next set of data while that operation is executing. If done correctly, VIP's vector unit ALUs can be kept busy every cycle, providing the peak computational throughput. This is different from a GPU, which requires each thread to also execute these overhead operations such as computing addresses, issuing loads and stores, and executing loops.

In order to keep hardware complexity low, VIP exposes the latency of the vector pipeline operations to the pro-

[2]The results presented in this work assume 16 bit integer/fixed-point data types; VIP provides a peak throughput of 1,280 GOp/s with these data types.

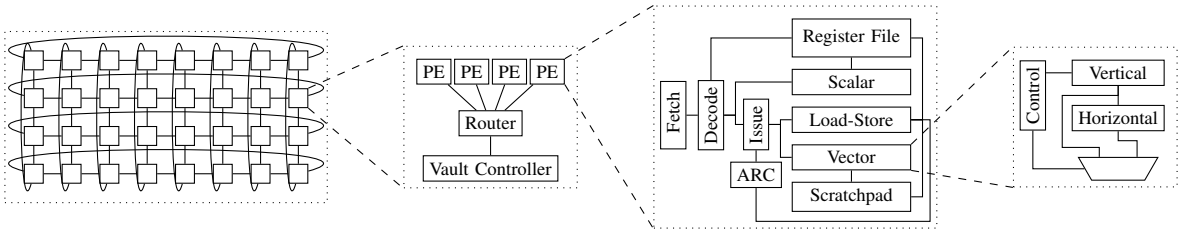


Figure 1: An overview of VIP's architecture, from the system level down to the datapath in a single PE. A Hybrid Memory Cube (HMC) has 32 vaults, which we connect via a 2D torus network. A vault has four processing engines (PEs) associated with it. A PE consists of unified fetch and decode, followed by independent vector, scalar, and load-store units. A unified issue unit is responsible for detecting data hazards in the scratchpad (used by vector and load-store pipelines) through an associative array lookup (ARC). The vector unit itself consists of a lightweight vertical unit for element wise operations, and a horizontal unit for reducing vectors to scalars. Both vertical and horizontal units have a 64 bit datapath which may operate on one 64 bit, two 32 bit, four 16 bit, or eight 8 bit data in a single cycle.

grammer or compiler, who may schedule instructions in order to improve utilization and avoid hazards. It also provides an instruction to drain the vector pipeline which may be used to conservatively avoid hazards in case such scheduling is not possible. Section III-B describes how we can avoid exposing the latency of vector operations at the cost of some hardware complexity. Additionally, VIP trades off support for predicated execution, vector masking, and precise exceptions for hardware simplicity. Table II lists VIP's instruction set.

B. VIP PE Microarchitecture

The VIP ISA provides a way of efficiently encoding the operations involved in the workloads considered. VIP's microarchitecture and memory system, however, are key to its performance.

To support f_6 category vector operations (matrix-vector operations), we need to provide efficient hardware support for reductions. A vector unit in VIP consists of a unit performing vertical (element wise) vector operations followed by a horizontal (reduction) vector unit. The reduction unit is bypassed for operations that do not require it. Both vertical and horizontal vector units have a 64 bit datapath which may perform one 64 bit, two 32 bit, four 16 bit, or eight 8 bit operations in a single cycle. Addition-like operations take one cycle for each ALU, multiplications require a 4-stage pipeline. If the vector's total footprint (number of elements times the size of an element) exceeds 64 bit, it is sent down the pipeline in multiple cycles, similar to the classic temporal vector-processing paradigm of early vector machines such as the CDC STAR-100 or the Cray-1 [44].

Section III-A discussed how VIP provides a 4 KiB SRAM scratchpad memory for its vector memory-memory processing paradigm. The scratchpad memory consists of eight banks each providing three read and two write ports, 8 bit wide. Swizzle logic is used to combine these ports into 64 bit ports; the banked structure allows the programmer to access any arbitrary address within the scratchpad without considering data alignment. Multiple read and write ports prevent port conflicts that would degrade performance; port conflicts are a concern with GPU shared memory. Two read ports and one write port are dedicated to the vector pipeline, and one read and one write port are dedicated to the load-store unit, allowing loads and stores to execute in parallel with vector operations.

VIP provides a scalar unit for executing basic control flow operations and for setting up addresses for data movement. The scalar unit has a reduced instruction set with a 64 bit datapath. The scalar register file contains 64 elements. This is on the higher end, but is useful as VIP does not have caches, which makes register spills very expensive, and because the system must also hold pointers within the scratchpad. In order to avoid hazards in the scalar pipeline, registers are augmented with a "valid" bit which is cleared whenever an instruction responsible for updating that register is issued and set upon completion of that instruction. Subsequent instructions accessing that register are stalled until the valid bit is set. In order to move data between the PE and DRAM, VIP provides a load-store unit which allows for 64 outstanding loads or stores to or from either scalar registers or scratchpad memory.

Instructions moving data between scratchpad and DRAM read source and destination pointers along with the length of data to be moved from scalar registers. This allows loading vector or matrix data with sizes that are known only at runtime. In order to detect hazards within the scratchpad, VIP provides an associative array, array range check (ARC) in Figure 1, which holds scratchpad start and end addresses upon the issue of an instruction to load data to the scratchpad. Any subsequent instructions accessing a region of scratchpad that overlaps with an ARC entry are stalled until the load completes and clears the ARC entry. The ARC has three read ports for two source and one destination address range, a write port to create an entry, and a clear port to delete an entry upon completion of a load. It has twenty entries, additional entries would require some careful RTL design to allow the ARC to work with the 0.8 ns clock cycle time. It is possible to also use the ARC to prevent hazards within the vector pipeline, freeing the programmer from scheduling instructions in order to prevent these hazards. This, however, will require increasing the size of the ARC and also result in additional ARC queries, which will in turn increase the power consumption.

VIP uses a unified front-end consisting of fetch and decode stages. Each VIP PE has a 1,024-entry instruction buffer that holds the program being executed. The decode

stage is responsible for sending data down the appropriate (vector, scalar, or load-store) pipeline. If any instruction is stalled in the decode/issue stages (e.g., due to a data dependency), all subsequent instructions are also stalled. We find that the in-order issue model provides us with close to peak performance. Instructions can, however, complete out of order. As we cannot efficiently checkpoint the state of the scratchpad, we choose to not support precise exceptions. We consider this an acceptable trade-off.

C. VIP Memory System

The workloads considered require a memory system that has high amounts of memory-level parallelism, and that can work efficiently with requests for short bursts of data. We therefore couple the VIP PEs with 3D stacked memory, organized in a manner similar to the HMC [22], with modest but key modifications which we will describe in this section. The memory system, like the HMC, is divided into 32 vertical partitions or vaults arranged in an 8×4 grid. Each vault contains 16 DRAM banks connected using through-silicon vias (TSVs). Banks within a vault share data TSVs but use independent control TSVs, so each bank is also a rank. A bank contains 65,536 rows, which in turn contain 256 B, each accessed as 32 B columns. Each vault provides a $10\,\mathrm{GB\,s^{-1}}$ DRAM bandwidth totaling $320\,\mathrm{GB\,s^{-1}}$ for the stack.

VIP's 128 PEs are distributed among the 32 vaults, four PEs in each vault. The vaults are connected via a 2D torus network, and PEs within a vault, along with the vault controller, are connected in a star topology. The network links are bidirectional, 64 bit links in each direction. With a 1.25 GHz clock, this configuration provides a $10\,\mathrm{GB\,s^{-1}}$ bandwidth on each network link, ensuring that the network does not become a bottleneck.

The default HMC address mapping scheme [22] indexes vaults using low address bits. This scheme exposes the most memory parallelism for a device accessing the HMC from outside. However, we want the PEs be able to access their local vaults and keep traffic on the on-chip network to a minimum. To accomplish this, we index the vaults using the most significant address bits; this allows PEs to safely access data within their vaults.

While there are advantages to being located within the memory stack for energy efficiency, and we argue that VIP is sufficiently general to be an excellent candidate for such integration, this may not be feasible from an economic perspective [51]. We do not assume any special characteristics within the memory stack (e.g., we do not assume that the bandwidth within the memory stack is more than the bandwidth outside, nor do we assume logic-in-memory requirements). Therefore, our simulation results discussed in Section VI won't change significantly if it is economically infeasible to include VIP within the HMC logic layer. Although we will suffer additional latency from the SERDES links, we can prefetch data more aggressively in software to hide this latency. The address interleaving discussed in the previous paragraph may be changed using a logical to physical address translation. This is simpler than virtual memory, as the mapping is known statically and involves shuffling some bits in memory requests.

The HMC uses a closed-page policy. Intuitively, this helps workloads that request an entire cache line of data. VIP forgoes caches due to the streaming nature of the BP application. (We find that we can achieve good performance on all benchmarks without the use of caches; we attribute this to the improved memory-level parallelism that provides high-bandwidth data access.) Given the absence of caches, an open-page policy provides lower DRAM access latency for multiple requests to nearby memory addresses. Our experiments (Section VI-C) confirm this intuition. VIP, therefore, uses an open-page policy.

Refresh in modern DRAM systems is a source of overhead. We observe that the JEDEC DDR4 standard allows for refresh to occur at a higher frequency and for a shorter duration (effectively decreasing both t_{REFI} and t_{RFC}) [1, 35]. In our experiments, discussed in Section VI-C, we find that reducing both t_{RFC} and t_{REFI} so that we refresh rows every 1.95 μs instead of every 7.8 μs (approximately matching the DDR4 refresh_4x mode) results in low refresh overhead.

IV. Software Design

In this section, we describe how we write parallel implementations of BP-M, and convolutional and fully-connected layers for VGG networks [48]. While we believe VIP would make a good compiler target due to its use of a vector-processing paradigm, for now, we write code in assembly. All benchmarks use 16 bit dynamic fixed point arithmetic.

A. Belief Propagation

The BP-M algorithm (discussed in Section II-A) imposes a strict sequential order for message updates in a given direction; parallelism exists in the orthogonal direction. It is important to partition data so that processing engines (PEs) access their local vaults most of the time and minimize traffic on the on-chip network. To achieve this, we divide the image into a square grid of rectangular tiles, as many tiles per side as the number of vaults. PEs within a vault work on the tile assigned to that vault and update messages within the tile, accessing only local data. Once the PEs finish updating messages within the tile, they copy messages at the tile boundary to their neighboring vaults and move to the next tile to be processed. We assign tiles to each vault such that each row or column of the grid contains tiles assigned to different vaults (ensuring that all PEs have data for every message update direction) and that adjacent tiles are assigned to vaults that are immediate neighbors in the physical layout of the memory system (ensuring that the minimal communication at tile boundaries is limited to just one network link). This scheme in fact limits communication to a ring connecting all the vaults in the memory system. We use full-empty synchronization variables in DRAM to synchronize producer-consumer PEs at tile boundaries. A

```
1 ld.sram [16-bit] r11, r7, r61; Load messages
2 ld.sram [16-bit] r12, r8, r61; r61 = vector length
3 ld.sram [16-bit] r13, r9, r61; r7-9 = DRAM addresses
4 v.v.add [16-bit] r11, r11, r12; Update message
5 v.v.add [16-bit] r11, r11, r13
6 v.v.add [16-bit] r11, r11, r14
7 m.v.add.min [16-bit] r14, r15, r11; r15 = Smoothness cost
      ↪ in SRAM
8 st.sram [16-bit] r10, r14, r61 ; r14 = DRAM address
```

Figure 2: VIP assembly code fragment for a min-sum BP message update

distributed barrier (written so that PEs access either their own vaults or immediate neighbors) is used to synchronize all PEs at the end of message updates in a given direction. Figure 2 shows a VIP assembly code fragment for a single min-sum belief propagation (BP) message update. Lines 4 to 6 execute Equation (1a), while Line 7 executes Equation (1b). The actual code is software pipelined to load data four iterations before it is used.

B. Convolutional Neural Networks

Convolutional neural networks (CNNs) are easier to parallelize as each output feature for each layer may be computed in parallel. We utilize the X-Y structure of the activations, and divide the inputs for each layer into a series of X-Y tiles, which are assigned to vaults within VIP in the corresponding X-Y locations. Tiles in the Z dimension are assigned to adjacent vaults in the X dimension. The general pattern for computation in any CNN follows a template. Load in as many $k \times k \times z$ filters into the scratchpad as possible, while being able to also store $(k+1) \times k \times z$ inputs. While applying the loaded filters to the $k \times k$ window of inputs, prefetch the next $1 \times k \times z$ column of inputs. When these selected filters have been applied to all the inputs assigned to that PE, load in the next set of filters and repeat the process. Code is written in a way that outputs from one layer are already in the right location to be consumed as inputs by the next layer. The length of filters in the Z dimension, however, can result in the filters being too large for the 4 KiB scratchpad. In this case, we tile filters into shards and distribute them across vaults. PEs within these vaults compute local partial convolutions, synchronize, then accumulate these partial results. PEs access input features from their local tiles multiple times as they apply the input features and write partial outputs, but access remote vaults just once to accumulate these partial results. As a result, communication once again is limited to mostly local vaults.

VGG-16 and VGG-19 networks [48] use $k = 3, z = 64$. The first layer is an exception, as the input consists of just three channels. In this scenario, we are able to fit in all 64 filters in the scratchpad of a single PE, so different PEs within a single vault operate on different regions of the tile assigned to that vault. Later convolutional layers in VGG networks have feature sizes that are a multiple of 64, we divide these across multiple vaults as discussed in the previous paragraph. The last convolutional layers, however, have very small feature sizes, at just 14×14. We only use half the vaults in VIP for these layers. We find that this still results in an acceptable runtime.

We merge pooling operations into the code phase which collects partial results, adds biases, and applies rectified linear unit (ReLU). For later pooling layers, we are unable to do this as we run out of scratchpad space, so we perform the pooling operation separately.

C. Multi-layer Perceptrons

Parallelizing multi-layer perceptrons (MLPs) is fairly straightforward. At the end of the last pooling layer, we are left with a $25{,}088 \times 1$ vector which is distributed in segments among the vaults at the very top of the memory system. We distribute tiles from the $4{,}096 \times 25{,}088$ weight matrix among all the vaults of the memory system. A fully-connected layer is executed in three passes. First, all PEs copy assigned segments of the input vector into their local vaults. Second, PEs in a vault compute partial products of their respective matrix tiles with their vector segments. Third, PEs in the vaults on the left side of the memory system accumulate partial products from vaults in the same row, add biases and perform the ReLU operation. Subsequent fully-connected layers are executed in a similar way, alternating the way data are moved, from the left-side vaults to the top vaults.

V. Experimental Setup

We evaluate VIP's performance using detailed execution-driven simulations. We validate the simulation model and estimate area and power through RTL synthesis of a VIP processing engine (PE) in TSMC 28 nm technology using a commercial standard-cell library provided by ARM (Section VII). We write CUDA code for a baseline BP-M implementation, executed on a Nvidia Titan X (Pascal) GPU, we use existing accelerator and GPU baselines [9, 17, 25, 40] for VGG networks.

A. Simulation Infrastructure

We use an execution-driven microarchitecture simulator which faithfully models all pipelines including stalls and contention on shared ports in the VIP PE. We ensure that the simulator models the same pipeline structure validated through a synthesizable RTL implementation of a VIP PE. The on-chip network is modeled as a 8×4 2D torus. Contention and bandwidth is modeled at all injection and ejection ports, and we assume that each router+link hop incurs 3 cycles of latency. We use DRAMSim2 [43], an open source DRAM simulator, to model the memory system.

We use timing parameters from Kim et al. [28] with some changes, such as changing the address mapping scheme, row-buffer policy, and refresh rate as discussed in Section III-C. Table III lists the key parameters used in the memory system simulation.

We run a single BP-M iteration on the simulator to obtain the number of cycles required for one iteration. We verify that the simulated code is correct by comparing its outputs against a reference C++ implementation. Similarly, we execute an independent tile from each VGG-16 convolutional layer and verify the results against a reference

Table III: Parameters used in memory simulation

Parameter				Value	Parameter		Value
HMC vaults				32	Banks per vault		16
HMC vault data width				32 bit	Burst length		8
Row buffer policy				open-page	Cmd queue depth		32
Address mapping				vault-row-bank-col	Trans queue depth		32
t_{CK}	0.8 ns	t_{RP}	13.75 ns	t_{CCD} 5 ns	t_{RCD}	13.75 ns	
t_{CL}	13.75 ns	t_{WR}	15 ns	t_{RAS} 27.5 ns	t_{RFC}	81.5 ns	
t_{REFI}	1.95 µs						

C++ implementation. An independent tile is a segment of the input features that does not share any resources (PEs, memory requests, or network bandwidth) with another tile. All independent tiles have the same amount of work, or we simulate the largest independent tile in a given layer. Simulating a single independent tile greatly reduces the simulation time without affecting simulation accuracy. As we cannot break down the fully-connected layers into independent tiles, we simulate the complete network.

B. Baseline Implementations

GPU Implementation of BP-M While there has been prior work on accelerating belief propagation (BP) on GPUs [18, 29, 61], such work has been done on older GPUs which makes simply reporting those numbers unfair to them. Additionally, these works make assumptions on the nature of smoothness costs (e.g., the finite truncated model), use message update schedules different from BP-M, and do not provide source code.

We write our own hand-optimized BP-M implementation. We tried using Tensorflow [2], but the resulting code was very slow. This is probably because each message update had to be expressed as a combination of Tensorflow primitive operations, which resulted in a very large number of nodes in the Tensorflow dataflow graph. In order to extract high performance from Tensorflow, we would have to write and compile native code either for the BP-M algorithm or for a subset of operations involved. Instead of writing Tensorflow kernels in CUDA for the BP-M algorithm, we might as well write a native CUDA implementation of BP-M, knowing that the performance of the native implementation will be at least as good as the implementation that we could write within Tensorflow utilizing our new custom operator. We use Nvidia profiling tools to tune our implementation on an Nvidia Titan X (Pascal) GPU. As we cannot efficiently write BP-M code in Tensorflow, we cannot run BP-M on the Google Cloud TPU either, without knowledge of the low-level details of the tensor processing unit (TPU).[3]

The Titan X utilizes the Pascal architecture from Nvidia and provides a peak memory bandwidth of 480 GB s^{-1} and a peak computational throughput of 11 TFLOPS [60]. This GPU has the required memory bandwidth and significantly higher compute capability than required for BP-M. We use shared memory on the GPU to store the smoothness costs as well as the intermediate results of computation as the latency of shared memory is much less than the latency of GPU DRAM memory. We also tune our code to minimize the number of bank conflicts in shared memory. We do not, however, tune the kernel so that it would work well for only a particular image size or number of labels, instead we supply these as parameters to the kernel; nor do we make any assumptions on the structure of the smoothness cost functions. This is a fair comparison as we do not make these assumptions for the simulated VIP hardware either.

[3]Jouppi et al. [26] have published the microarchitecture of the TPU, consisting of a systolic array for multiply-accumulate (MAC) operations, so the TPU cannot in principle perform the min-sum BP update anyway.

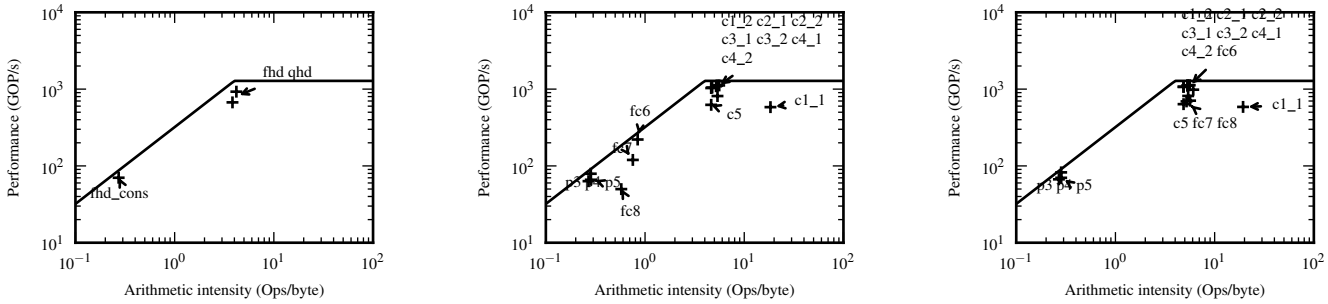
Reference CNN Architectures There are a number of existing accelerator implementations for convolutional neural networks (CNNs), e.g., Cambricon [33], Eyeriss [9], Neurocube [27], Neurostream [7], and Tetris [17]. Unfortunately, many of these implementations use different neural networks which makes direct comparison difficult. Eyeriss is a popular, widely-cited architecture which reports absolute performance on VGG-16 network, making it a suitable baseline. Tetris builds on Eyeriss by significantly increasing the number of PEs compared to Eyeriss, and using a Hybrid Memory Cube (HMC) to supply high bandwidth to these PEs. Unfortunately, Tetris only reports speedups with respect to a 2D design with LPDDR DRAM, which is sufficiently different from Eyeriss to render making direct performance comparisons impossible. Finally, GPU reference implementations of various CNNs are readily available and benchmarked on various GPU platforms. [25, 40]

VI. Evaluation

A. End-to-end Application Performance

Table IV shows an overview of the performance of the various workloads considered on VIP as well as on various baselines considered, including Nvidia Titan X (Pascal), Jetson TX2 and Volta GPUs, Eyeriss [9], Tile-BP [10] and Optical Gibbs' Sampling [55]. From the table, we see that VIP is faster than the Titan X for BP-M, the application for which VIP was originally designed. Additionally, we see that VIP's performance on convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs) exceeds the performance of Eyeriss, and is competitive with the Nvidia Jetson TX2 GPU. We will discuss these results in greater detail in following paragraphs.

Figure 3 shows where the applications considered (various belief propagation (BP) kernels and CNN and MLP layers from VGG-16 and VGG-19 networks) fall under VIP's performance roofline. The "roofline" represents the peak achievable performance of a system. The distance of a kernel below the roofline indicates the gap between the achieved performance and the theoretical peak performance. The x-axis shows the arithmetic intensity (in terms of number of operations performed for each byte of memory moved), while the y-axis shows the performance in GOp/s. Kernels at the roofline with low arithmetic intensity (to the left of the knee point in the roofline) are likely to be memory bound, kernels with high arithmetic intensity (to the right



(a) Belief propagation (b) VGG-16, batch size 1 (c) VGG-16, batch size 16

Figure 3: Roofline plots for VIP running belief propagation on a full-HD (fhd) and quarter-HD (qhd) image, and convolutional neural network (c), pooling (p) and multi-layer perceptron (fc) layers from VGG-16 [48] network for batch sizes of 1 and 16.

Table IV: A summary of the performance of this work—VIP—compared against prior work on ASIC accelerators and GPUs for MRF and CNN workloads. Asterisks (*) indicate that the systems described use a different algorithm to solve the same problem.

Markov random fields					
System	**Iterations**	**Time (ms)**	**Power (W)**	**Tech (nm)**	**Area (mm^2)**
Optical Gibbs' Sampling [55]	5000*	1,100	12	15	200 + 12
Tile BP (720p) [10]	(1,2)*	32.7	0.242	90	9
Pascal Titan X	8	92.2	250	16	471
VIP (baseline BP-M)	**8**	**41.3**	**3.5**	**28**	**18**
VIP (hierarchical BP-M)	**5**	**36.3**	**3.5**	**28**	**18**
Convolutional neural networks—VGG-16 (convolution layers only)					
System	**Batch size**	**Time (ms)**	**Power (W)**	**Tech (nm)**	**Area (mm^2)**
Eyeriss [9]	3	4,309	0.236	65	12
VIP	**3**	**91.6**	**4.8**	28	18
Convolutional neural networks—VGG-16 (full network)					
System	**Batch size**	**Time (ms)**	**Power (W)**	**Tech (nm)**	**Area (mm^2)**
Pascal Titan X [25]	16	41.6	250	16	471
VIP	**16**	**492.4**	**4.8**	28	18
VIP	**1**	**32.3**	**4.8**	**28**	**18**
Convolutional neural networks—VGG-19 (full network)					
System	**Batch size**	**Time (ms)**	**Power (W)**	**Tech (nm)**	**Area (mm^2)**
Volta [13, 40]	1	2.2	144	12	815
Jetson TX2 [40]	1	42.2	10	16	unknown
VIP	**1**	**40.6**	**4.8**	**28**	**18**

of the knee point in the roofline) will likely be compute bound. Due to the fact that VIP supports scalar and vector operations, and that the system can perform more operations with smaller data types, we must be careful when defining performance. In these plots, we define performance as only the number of 16 bit ALU operations performed by the vector units, but we include memory accesses by the scalar pipeline (e.g., for synchronization) when reporting arithmetic intensity.

Belief Propagation We run a baseline as well as a hierarchical BP-M algorithm in our simulator, on full-HD images with 16 labels. (The hierarchical BP-M algorithm is similar to the one proposed by Felzenszwalb and Huttenlocher [15].) The baseline BP-M is as discussed in Section II-A, while the hierarchical implementation consists of four phases—construct a coarser version of the graph by pooling neighboring data costs, perform BP-M on the resulting quarter-HD Markov random field (MRF), copy messages back to the original full-HD MRF, then perform BP-M on the full-HD MRF. Hierarchical BP-M requires fewer iterations as it converges faster than baseline BP-M.

A single iteration of BP-M takes 5.2 ms, eight BP-M iterations require 41.3 ms, allowing VIP to execute a depth-from-stereo task at a real-time frame rate of 24 fps on full-HD video with good results. The construct and copy operations for hierarchical BP-M require 0.36 ms and 1.26 ms, respectively, while an iteration of BP-M on the quarter-HD MRF requires 1.8 ms. As construct and copy are performed only once per frame, five iterations of hierarchical BP-M will require 36.3 ms, which again allows VIP to process full-HD video at 24 fps. As a reference, the Nvidia Titan X GPU requires 11.5 ms for a single BP-M iteration, not counting the time taken to move data between DRAM and GPU memory. Eight iterations will therefore require 92.2 ms. The Nvidia profiler reports that the GPU performance is limited by both instruction and memory latency. We attribute this to the fact that the BP-M algorithm, while highly parallel, does not have sufficient parallelism to keep the GPU fully occupied. On the other hand, Tile-BP [10] requires 32.7 ms for a 720p image. Tile-BP only stores messages at tile boundaries, recomputing messages within a tile, reducing the storage requirements to fit within on-chip SRAM. It, however, can perform only one effective BP-M iteration against VIP's eight. Optical Gibbs' sampling [55] uses a different algorithm, Gibbs' sampling, instead of BP for the same application (MRF labeling). As Gibbs' sampling requires more iterations to converge, their work is predicted to require 5000 iterations and 1100 ms, based on projections of a future technology.

The roofline plot in Figure 3a shows that the BP kernels (full and quarter HD) lie near the knee point. This indicates that VIP balances the memory bandwidth and compute required for the BP kernel, not surprising given this was a design objective. The construct operation simply adds four vectors, hence its arithmetic intensity is low. Even so, it is near the roofline indicating that we achieve close to peak

performance for a kernel bound by memory bandwidth. This analysis also indicates that the Nvidia Jetson GPU will be severely bottlenecked by its 60 GB s^{-1} memory bandwidth.

Convolution layers We run individual layers from VGG-16 and VGG-19 [48] networks on our VIP simulator. The convolution, rectified linear unit (ReLU) and pooling layers before the first fully-connected layer (`fc6`) require a total of 30.9 ms for VGG-16 and 39.2 ms for VGG-19 when operating with a batch size of 1. With a batch sizes of 3 and 16, VGG-16 convolution layers take 91.6 ms and 488 ms respectively. The linear relationship between batch size and execution time shows that VIP achieves good performance without the need for batching.

As a comparison, Eyeriss [9] reports requiring 4309 ms for VGG-16 convolution layers with a batch size of 3. We note that Eyeriss uses a different technology node (65 nm v. 28 nm) and has different silicon area (12 mm^2 v. 18 mm^2) than VIP. We therefore, try and normalize Eyeriss' resources against VIP to make a meaningful comparison. In order to normalize area and technology, we divide Eyeriss' runtime by $^{18}/_{12}$ to normalize area, and by $(^{65}/_{28})^2$ to normalize technology. We note that Eyeriss operates at 200 MHz while VIP operates at 1.25 GHz. We optimistically assume that Eyeriss will be able to achieve a 1.25 GHz clock speed, and that its performance would scale linearly with area and clock speed without any other bottlenecks such as DRAM bandwidth. We must therefore, divide Eyeriss' runtime by another $^{25}/_{4}$ to adjust for clock speed. The end result of this analysis is that VIP is less than 10 % worse than Eyeriss-scaled, at Eyeriss' own and only game.

Figure 3b shows various CNN kernels on a roofline plot. We notice that the pooling layers are memory-bound, but very close to the roofline. The convolution layers lie near the knee point of the roofline plot, indicating once again that VIP is well-balanced for convolutional layers, which account for a bulk of the execution time. There are some exceptions, however. The first convolutional layer (`c1_1`) is different from the other convolutional layers, as discussed in Section IV-B. The ability to load all filters into the scratchpad means that more arithmetic operations are performed on features every time they are loaded, increasing arithmetic intensity. The small vector lengths, however, mean that the data are processed quickly, which exposes latency of the control code (loops, data-movement). This prevents VIP from achieving peak performance for this layer. On the other hand, the later convolutional layers (`c5`) suffer in performance because the input feature maps are very small, leading to small tile sizes distributed across only half the vaults, decreasing both memory bandwidth and computational capacity.

Fully connected layers We also run fully-connected layers from VGG-16 and VGG-19 networks [48]. (The two networks use the same fully-connected layers.) When running with a batch size of 1, the fully-connected layers take 1.4 ms, with a batch size of 3, they take 1.8 ms, and with a batch size of 16, take 4.4 ms. Figures 3b and 3c show where the fully-connected layers lie under the performance roofline of VIP. With a batch size of 1, the first fully-connected layer (`fc6`) lies near on the roofline, but both arithmetic intensity and peak performance drop as the weight matrix gets smaller and data-movement and synchronization overheads increase for later layers (`fc7` and `fc8`). On increasing the batch size to 16, these overheads are reduced, increasing the arithmetic intensity. The fully-connected layers now lie near the knee point (Figure 3c).

VGG performance The complete VGG-16 and VGG-19 networks (convolutional and fully-connected layers) require 32.3 ms and 40.6 ms respectively with a batch size of 1. With a batch size of 16, VIP requires 492 ms on VGG-16 network. The Nvidia Titan X (Pascal) GPU requires 41.6 ms on the same network with the same batch size [25]. Nvidia reports that the Nvidia Volta GPU requires 2.2 ms for VGG-19 network with a batch size of one, using its special Tensor cores which are optimized for fast matrix-matrix multiplications [40]. While these Tensor cores cannot accelerate the min-sum BP algorithm, they are useful for accelerating CNN with high arithmetic intensity. The Volta occupies 815 mm^2 in 12 nm technology, a similar scaling analysis as with Eyeriss indicates that this is $\sim 250\times$ VIP's area. The Nvidia Jetson TX2, on the other hand, requires 42.2 ms for VGG-19 network, again with a batch size of one [40]. It is important to note that unlike these GPUs which require batching to achieve peak frame rate, VIP achieves very close to its peak frame rate with no batching, making it suitable for real-time systems prioritizing latency over throughput.

B. Sensitivity to Architectural Choices

VIP differs from a traditional vector processor in two distinct ways that make it better suited for BP: its vector unit is designed to pipeline vertical and horizontal vector operations, and it uses a scratchpad instead of a traditional vector register file. In order to evaluate the benefits of these modifications, we write code within VIP to restrict these features and emulate a traditional vector register machine. We achieve this by not using the reduction unit and by restricting the location of vectors within the scratchpad to sixteen 256 B locations, a similar configuration to IBM's Active Memory Cube [37]. We assume that the baseline vector ISA provides instructions that can either extract or insert a scalar value from or to an arbitrary location in a vector. We further assume that a segment of a vector can be shuffled between two different vector registers. Moving N elements between vector registers will require $\lceil N/w \rceil$, if w is the number of elements that can be read or written from the register file port. Similarly, extracting a scalar from or inserting it into an arbitrary location in a vector register requires one cycle. We write code using four configurations—baseline VIP with its scratchpad and reduction units (SP+R), with a scratchpad but not using any reduction instructions (SP-R), with a register file and a reduction unit that writes

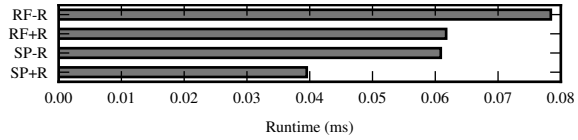
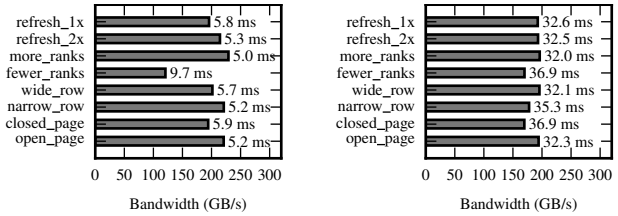


Figure 4: Execution time for BP-M updates in the vertical direction for a 64×32 tile on four configurations—VIP with its scratchpad and reduction units (SP+R), with a scratchpad without using the reduction units (SP-R), a register file with a reduction unit (RF+R), and a register file without reduction units (RF-R).



(a) Belief propagation (b) CNN VGG-16

Figure 5: Memory bandwidth (and execution time) for BP on a full-HD image and VGG-16 [48] network, end-to-end (including the convolution, ReLU, pooling, and fully-connected layers) under various memory configurations obtained by tweaking the one in Table III.

its results into a scalar register (RF+R), and with a register file without a reduction unit (RF-R). In order to maximize the utilization of the vector register file, we pack multiple (8) 32 B vectors within the 256 B vector register, unpacking data to be operated upon, and repacked for storage. We simulate BP-M on a 64×32 tile, a size close to the tile size used when working on a full-HD image. We update messages in the vertical direction, and store messages and data costs such that eight vectors may be loaded into the vector register file using a single contiguous load operation. When simulating a configuration with a scratchpad instead of a vector register file, we load each vector individually. This setup provides the maximum possible advantage to the register file configuration by reducing the number of outstanding loads in flight and by reducing the address bandwidth. Figure 4 shows the results of this comparison. We see that configurations without the reduction unit require a longer execution time than a comparable configuration with a reduction unit as the reduction unit doubles the number of operations that may be performed in each cycle. We also observe that configurations with a traditional register file perform worse than configurations with a scratchpad. This is because unpacking and packing registers adds overheads increasing execution time.

C. Sensitivity to Memory Parameters

We start with the configuration shown in Table III ("open_page"). We change the row-buffer policy to closed page ("closed_page"). To simulate the effect of varying memory parallelism, we start with the open_page policy, and reduce and increase the number of ranks (the Hybrid Memory Cube (HMC) has one bank per rank, so these terms are used interchangeably) by 4× (we increase and decrease the number of rows per bank to keep the DRAM size constant) ("fewer_ranks" and "more_ranks"). We study the effect of wider and narrower rows by increasing and decreasing the width of the DRAM rows (decreasing and increasing the number of rows) by 4× ("wide_row" and "narrow_row"). Finally, we start with the open_page policy (which implicitly uses the refresh_4x mode described in the JEDEC DDR4 standard [1]) and study the effect of increasing both t_{RFC} and t_{REFI} by 2× and 4× ("refresh_2x" and "refresh_1x"). Figure 5 shows the achieved memory bandwidth (and execution time) for the end-to-end benchmarks—one iteration of full-HD BP-M and the entire VGG-16 network under these configurations.

A closed page policy instead of an open page policy has a detrimental effect on the effective memory bandwidth, which confirms our intuition in Section III-C. If we decrease the number of ranks in the system, the memory bandwidth drops and the execution time increases. This is not surprising—increased memory-level parallelism allows more DRAM requests to be in flight at any time, increasing the achievable bandwidth. Increasing the refresh interval (and consequently the refresh cycle time) significantly decreases memory bandwidth and increases execution time for BP, while CNNs are affected to a much lesser degree. We attribute this to the fact that CNNs access memory through few, large requests, while BP accesses memory through many small requests. As a result, BP is more sensitive to memory system stalls. Similarly, CNNs show higher bandwidth with wider rows, while BP shows a higher bandwidth with narrow rows. We again attribute this to the nature of DRAM requests. Narrow rows result in data being scattered across multiple rows, necessitating the row to be closed and reopened when another processing engine (PE) requests the same data when executing CNNs.

VII. RTL Synthesis

We synthesize a single VIP processing engine (PE) in TSMC 28 nm technology using a commercial standard-cell library from ARM in order to estimate VIP's area and power. We use CACTI 6.5 [36] to estimate the area and energy required by SRAMs. (The smallest feature size that CACTI can model is 32 nm, so the area and power estimates provided by CACTI will be somewhat pessimistic.) In order to make the final design as close as possible to a layout with real SRAMs, we use the area and energy numbers from CACTI to create black-box SRAM models for the ASIC toolchain. We use eight 512×8 bit SRAMs for the scratchpad, a 64×64 bit SRAM for the register file, a 64×32 bit SRAM for the load-store queue, and a 1024×32 bit SRAM for the instruction buffer. The synthesized RTL code meets the 0.8 ns clock period assumed in our simulations. We run small belief propagation (BP) and convolutional neural network (CNN) kernels on our RTL model and verify the results of the RTL simulation against a reference implementation to check for correctness. We use switching activity factors from these RTL simulations as inputs to Synopsys PrimeTime to estimate power consumption.

Figure 6 shows the layout of a single PE after place-and-route. A single VIP PE requires 0.141 mm^2 of silicon

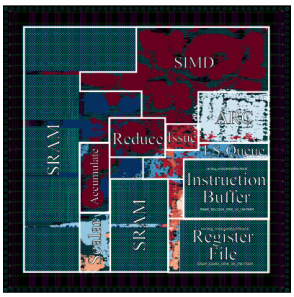


Figure 6: The generated layout of one VIP processing engine (PE).

area, and consumes 27 mW and 38 mW of power when executing BP and CNN kernels respectively. (CNNs require more power than BP, as they use multipliers). 128 PEs will, therefore, occupy 18 mm^2 in area and consume 3.5 W to 4.8 W of power, in addition to the power consumed by the 3D-stacked DRAM. An early prototype of the Hybrid Memory Cube (HMC) in 50 nm technology was found to consume 10 $pJ\,bit^{-1}$ [24]. At 320 $GB\,s^{-1}$, this HMC would require 25.6 W of power. IBM, on the other hand, estimates that a 320 $GB\,s^{-1}$ HMC will require only 5 W of power in 14 nm technology [23]. Azarkhish et al. [6] synthesize a HMC vault controller and estimate that each vault controller takes 0.62 mm^2, or 19.84 mm^2 for 32 vault controllers. The size of a 16-vault HMC DRAM die is 68 mm^2 [24], the size of a 32 vault die will be even larger. This would suggest that VIP could be integrated into the logic die of a 3D-stacked memory system like the HMC, although placing VIP outside the 3D stack will not significantly affect performance, as discussed in Section III-C.

VIII. Related Work

ASIC implementations Prior work involving ASIC for belief propagation (BP) includes a tile-based BP algorithm [10, 29], which stores only messages at the edge of a tile, using multiple BP-M iterations to recompute messages within a tile, achieving 30 fps for 720p video, but performing only one effective iteration. Work by Tseng and Chang [52] uses a "spinning message" update, based on the message storage scheme for bipartite graphs proposed by Felzenszwalb and Huttenlocher [15]. ASIC implementations for convolutional neural networks (CNNs) include Eyeriss [9] and Tetris [17], which have been discussed in previous sections. Other work includes Neurostream [7] and Neurocube [27]. Both these works involve multiply-accumulate (MAC) units on the logic die of a Hybrid Memory Cube (HMC), supplied data by finite state machines that generate address request streams to DRAM. Cnvlutin [3] saves computation by skipping multiplications where one of the operands is a zero. SCNN [41] goes further by storing the weights and activations in a sparse-compressed format. Cambricon [33] presents itself as an ISA for neural networks. Similarly, Google's tensor processing unit (TPU) deployed in its datacenters uses a systolic array of MAC operations. Both Cambricon and the TPU have been discussed in Section II-D. PuDianNao [32] is a machine learning accelerator for multiple machine learning kernels offering a pipelined, fixed-function implementation of parallel and reduction operations. VIP, on the other hand, leaves the choice of these operations to the programmer thereby providing more flexibility than a datapath with fixed functional units. Spert-II [56], developed in the 90s, adds vector instructions to the MIPS ISA for training and inference on neural networks. It uses 16 bit vector elements, and was 30 times faster than a SPARC-20 workstation at the time. ScaleDeep [53] allows the execution of neural networks at scale, such as in a datacenter. ScaleDeep's approach is a modular one—providing both compute-heavy and memory-heavy tiles which are optimized for their respective tasks, which are composed to create a big system. SODA [31] and AnySP [58] also have combinations of scalar and SIMD processors for signal processing. Their design, however, is tailored to work with very wide vectors minimizing latency. VIP, on the other hand, works with short vectors, providing more processing engines (PEs), and using the long latency of vector operations to hide memory accesses. Unlike IBM's Active Memory Cube [37], VIP does not force the PEs in a vault to operate in lock step via a VLIW model either, allowing VIP to also operate on irregular graphs.

FPGA implementations Prior work involving FPGA implementations for BP include work by Park et al. [42] that uses 320 parallel processing engines spread across two Xilinx Virtex-II FPGAs. Choi and Rutenbar [12], later extended [21] use TRW-S, an alternate form of BP, on a Convey HC-1 platform. Lin et al. [30] use the block RAMs (BRAMs) inside the FPGA to provide high-bandwidth memory. Their solution, however, does not work for graphs with loops and therefore cannot be used for vision applications. Additionally, it is limited by the size of BRAMs inside the FPGA. Venice [45] is a soft FPGA-based vector processor. As a soft accelerator on an FPGA, Venice runs at a low clock frequency and therefore cannot provide the necessary throughput. Wei et al. [57] use systolic array for CNNs that achieves comparable performance against VIP. Nurvitadhi et al. [39] and Zhao et al. [62] employ binarized neural networks, i.e. neural networks where weights are expressed as binary values instead of as real numbers, and map these to FPGA platforms. Aydonat et al. [5] use OpenCL to map deep neural networks (DNNs) to an FPGA platform, their work uses special Winograd transformations to reduce the computational complexity of the convolution operation. ESE [19] is an FPGA system for sparse long short term memory (LSTM) networks. Tabla [34] is system that can generate FPGA accelerators for a class of machine learning algorithms that employ stochastic gradient descent, while DnnWeaver [47] generates FPGA implementations for DNNs. Tabla focuses on training, not inference, while DnnWeaver works on DNNs, not BP. Brainwave [14] stores neural network parameters within the BRAMs within FPGAs, using datacenter-scale deployment for large models. Brainwave presently does not support BP. Even if support for BP were added

to Brainwave, this model may be inefficient for very large data sizes, e.g., BP-M on a full-HD image requires over 300 MiB of storage.

Non-conventional devices A number of works have proposed using ReRAM crossbar arrays for dense matrix multiplication in DNNs [8, 11, 46, 49]. These ReRAM crossbars can only perform matrix multiplication by summing currents, they cannot perform the min-reduction in Equation (1b) required for BP. Optical Gibbs' sampling [55], discussed earlier, proposes using optical resonant units shrunk to fit multiple on a single GPU. VIP uses CMOS technology and can be deployed right away, while ReRAMs and optical resonant units are exotic devices and their results are based on projections of a future technology.

IX. Conclusions

We have presented VIP (Versatile Inference Processor), a highly programmable architecture for machine learning inference algorithms, including belief propagation (BP) on probabilistic graphical models (PGMs) and deep neural networks (DNNs) such as convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs). Through detailed execution-driven simulations backed by RTL synthesis, we have shown that VIP can achieve online, real-time vision throughput (24 fps, batch size of one) across these classes of inference workloads at low power consumption. Our RTL synthesis of a VIP processing engine (PE) in TSMC 28 nm technology, using a commercial standard-cell library supplied by ARM, results in 18 mm^2 of silicon area and 3.5 W to 4.8 W of power consumption for all 128 VIP PEs combined.

Acknowledgments

This work was supported by AFOSR grant FA9550-5-1-0311 and by a contract with Cavium. We are grateful to Avinash Sodani for his generous feedback.

References

[1] JC-42.3C. *DDR4 SDRAM*. Tech. rep. JESD79-4B. JEDEC, June 2017.

[2] M. Abadi et al. "TensorFlow: a system for large-scale machine learning." In: *Operating systems design and implementation*. USENIX Association, 2016.

[3] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. "Cnvlutin: ineffectual-neuron-free deep neural network computing." In: *Intl. symp. on computer architecture*. 2016.

[4] *ARM architecture reference manual supplement: the scalable vector extension (SVE), for ARMv8-A (beta)*. Tech. rep. DDI 0584A.a. ARM, 2017.

[5] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu. "An OpenCL deep learning accelerator on Arria 10." In: *Intl. symp. on field-programmable gate arrays*. 2017.

[6] E. Azarkhish, C. Pfister, D. Rossi, I. Loi, and L. Benini. "Logic-base interconnect design for near memory computing in the smart memory cube." In: *IEEE trans. on VLSI systems* 25.1 (Jan. 2017), pp. 210–223.

[7] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. "Neurostream: scalable and energy efficient deep learning with smart memory cubes." In: *IEEE trans. on parallel and distributed systems* 29.2 (Feb. 2018), pp. 420–434.

[8] M. N. Bojnordi and E. Ipek. "Memristive Boltzmann machine: a hardware accelerator for combinatorial optimization and deep learning." In: *Intl. symp. on high performance computer architecture*. 2016.

[9] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. "Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks." In: *IEEE j. of solid-state circuits* 52.1 (Jan. 2017), pp. 127–138.

[10] C.-C. Cheng, C.-T. Li, C.-K. Liang, Y.-C. Lai, and L.-G. Chen. "Architecture design of stereo matching using belief propagation." In: *Intl. symp. on circuits and systems*. 2010.

[11] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. "PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory." In: *Intl. symp. on computer architecture*. 2016.

[12] J. Choi and R. Rutenbar. "Hardware implementation of MRF MAP inference on an FPGA platform." In: *Intl. conf. on field-programmable logic and applications*. 2012.

[13] J. Choquette. "Nvidia's Volta GPU. Programmability and performance for GPU computing." Hotchips. 2017.

[14] E. Chung et al. "Accelerating persistent neural networks at datacenter scale." Hotchips. 2017.

[15] P. F. Felzenszwalb and D. P. Huttenlocher. "Efficient belief propagation for early vision." In: *Intl. j. of computer vision* 70.1 (May 2006), pp. 41–54.

[16] M. Fishelson and D. Geiger. "Exact genetic linkage computations for general pedigrees." In: *Bioinformatics* 18 Suppl 1 (2002), S189–98. PMID: 12169547.

[17] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis. "TETRIS: scalable and efficient neural network acceleration with 3D memory." In: *Intl. conf. on architectural support for programming languages and operating systems*. 2017.

[18] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan. "GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction." In: *Workshop on pattern recognition in remote sensing*. 2008.

[19] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al. "ESE: efficient speech recognition engine with sparse LSTM on FPGA." In: *Intl. symp. on field-programmable gate arrays*. 2017.

[20] D. E. Heckerman, E. J. Horvitz, and B. N. Nathwani. "Toward normative expert systems: part I. the pathfinder project." In: *Methods of information in medicine* 31.2 (June 1992), pp. 90–105. PMID: 1635470.

[21] S. Hurkat, J. Choi, E. Nurvitadhi, J. F. Martínez, and R. A. Rutenbar. "Fast hierarchical implementation of sequential tree-reweighted belief propagation for probabilistic inference." In: *Intl. conf. on field-programmable logic and applications*. 2015.

[22] Hybrid Memory Cube Consortium. *Hybrid memory cube specification 2.1*. Tech. rep. HMC-30G-VSR PHY. Oct. 2015.

[23] A. C. Jacob et al. *Compiling for the Active Memory Cube*. Tech. rep. RC25644 (WAT1612-008). IBM Research Division, Dec. 2016.

[24] J. Jeddeloh and B. Keeth. "Hybrid memory cube: new DRAM architecture increases density and performance." In: *Symp. on VLSI technology*. 2012.

[25] J. Johnson. *CNN benchmarks*. URL: https://github.com/jcjohnson/cnn-benchmarks (visited on 11/12/2017).

[26] N. P. Jouppi et al. "In-datacenter performance analysis of a tensor processing unit." In: *Intl. symp. on computer architecture*. 2017.

[27] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. "Neurocube: a programmable digital neu-

romorphic architecture with high-density 3D memory." In: *Intl. symp. on computer architecture*. 2016.

[28] G. Kim, J. Kim, J. H. Ahn, and J. Kim. "Memory-centric system interconnect design with hybrid memory cubes." In: *Intl. conf. on parallel architectures and compilation techniques*. 2013.

[29] C.-K. Liang, C.-C. Cheng, Y.-C. Lai, L.-G. Chen, and H. H. Chen. "Hardware-efficient belief propagation." In: *IEEE trans. on circuits and systems for video technology* 21.5 (May 2011), pp. 525–537.

[30] M. Lin, I. Lebedev, and J. Wawrzynek. "High-throughput Bayesian computing machine with reconfigurable hardware." In: *Intl. symp. on field-programmable gate arrays*. 2010.

[31] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. "SODA: a low-power architecture for software radio." In: *Intl. symp. on computer architecture*. 2006.

[32] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. "PuDianNao: a polyvalent machine learning accelerator." In: *Intl. conf. on architectural support for programming languages and operating systems*. 2015.

[33] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. "Cambricon: an instruction set architecture for neural networks." In: *Intl. symp. on computer architecture*. 2016.

[34] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. "TABLA: a unified template-based framework for accelerating statistical machine learning." In: *Intl. symp. on high performance computer architecture*. 2016.

[35] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez. "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems." In: *Intl. symp. on computer architecture*. 2013.

[36] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. *CACTI 6.0: a tool to model large caches*. Tech. rep. HPL-2009-85. HP Laboratories, Apr. 2009.

[37] R. Nair et al. "Active memory cube: a processing-in-memory architecture for exascale systems." In: *IBM j. of research and development* 59.2/3 (Mar. 2015), 17:1–17:14.

[38] L. M. Ni and A. K. Jain. "A VLSI systolic architecture for pattern clustering." In: *IEEE trans. on pattern analysis and machine intelligence* PAMI-7.1 (Jan. 1985), pp. 80–89.

[39] E. Nurvitadhi et al. "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" In: *Intl. symp. on field-programmable gate arrays*. 2017.

[40] *Nvidia deep learning platform. Technical overview*. 2017.

[41] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. "SCNN: an accelerator for compressed-sparse convolutional neural networks." In: *Intl. symp. on computer architecture*. 2017.

[42] S. Park, C. Chen, and H. Jeong. "VLSI architecture for MRF based stereo matching." In: *Embedded computer systems: architectures, modeling and simulation*. 2007.

[43] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. "DRAMSim2: a cycle accurate memory system simulator." In: *IEEE computer architecture letters* 10.1 (Jan. 2011), pp. 16–19.

[44] P. B. Schneck. *Supercomputer architectures*. Ed. by D. DeGroot. Kluwer Academic Publishers, 1987.

[45] A. Severance and G. Lemieux. "VENICE: a compact vector processor for FPGA applications." In: *Intl. conf. on field-programmable technology*. 2012.

[46] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. "ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars." In: *Intl. symp. on computer architecture*. 2016.

[47] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. "DnnWeaver: from high-level deep neural models to FPGAs." In: *Intl. symp. on microarchitecture*. 2016.

[48] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition." In: *ArXiv e-prints* (2014). arXiv: 1409.1556.

[49] L. Song, X. Qian, H. Li, and Y. Chen. "PipeLayer: a pipelined ReRAM-based accelerator for deep learning." In: *Intl. symp. on high performance computer architecture*. 2017.

[50] M. F. Tappen and W. T. Freeman. "Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters." In: *Intl. conf. on computer vision*. Vol. 2. 2003.

[51] J. Torrellas. "FlexRAM: toward an advanced intelligent memory system: a retrospective paper." In: *Intl. conf. on computer design*. 2012.

[52] Y.-C. Tseng and T.-S. Chang. "Architecture design of belief propagation for real-time disparity estimation." In: *IEEE trans. on circuits and systems for video technology* 20.11 (Nov. 2010), pp. 1555–1564.

[53] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, et al. "ScaleDeep: a scalable compute architecture for learning and evaluating deep networks." In: *Intl. symp. on computer architecture*. 2017.

[54] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. "Show and tell: a neural image caption generator." In: *ArXiv e-prints* (Nov. 2014). arXiv: 1411.4555.

[55] S. Wang, X. Zhang, Y. Li, R. Bashizade, S. Yang, C. Dwyer, and A. R. Lebeck. "Accelerating Markov random field inference using molecular optical Gibbs sampling units." In: *Intl. symp. on computer architecture*. 2016.

[56] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. "Spert-II: a vector microprocessor system." In: *Computer* 29.3 (Mar. 1996), pp. 79–86.

[57] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs." In: *Design automation conf.* 2017.

[58] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. "AnySP: anytime anywhere anyway signal processing." In: *Intl. symp. on computer architecture*. 2009.

[59] Y. Wu et al. "Google's neural machine translation system: bridging the gap between human and machine translation." In: *ArXiv e-prints* (2016). arXiv: 1609.08144.

[60] M. Wuebbling. *The new NVIDIA TITAN X: the ultimate. period.* July 21, 2016. URL: https://blogs.nvidia.com/blog/2016/07/21/titan-x/ (visited on 09/26/2017).

[61] X. Xiang, M. Zhang, G. Li, Y. He, and Z. Pan. "Real-time stereo matching based on fast belief propagation." In: *Machine vision and applications* 23 (2012), pp. 1219–1227.

[62] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. B. Srivastava, R. Gupta, and Z. Zhang. "Accelerating binarized convolutional neural networks with software-programmable FPGAs." In: *Intl. symp. on field-programmable gate arrays*. 2017.