TCX: A Programmable Tensor Processor

Tailin Liang[§†¶], Lei Wang[§¶], Shaobo Shi[§†], John Glossner[§‡], and Xiaotong Zhang[*§]

[¶] Equal Contribution, [*] Corresponding Author

[§] School of Computer Science and Communication Engineering, University of Science and Technology, Beijing 100083, China

Email: tailin.liang@xs.ustb.edu.cn, {wanglei, jglossner, zxt}@ustb.edu.cn

[†] Hua Xia General Processor Technologies, Beijing 100080, China

[‡] General Processor Technologies, Tarrytown, NY 10591, United States

***Abstract*—Neural network processors and accelerators are domain-specific architectures deployed to solve the high computational requirements of deep learning algorithms. This paper proposes a new instruction set extension for tensor computing, TCX, with RISC-style instructions and variable length tensor extensions. It features a multi-dimensional register file, dimension registers, and fully generic tensor instructions. It can be seamlessly integrated into existing RISC ISAs and provides software compatibility for scalable hardware implementations. We present an implementation of the TCX tensor computing accelerator using an out-of-order microarchitecture implementation. The tensor accelerator is scalable in computation units from several hundred to tens of thousands. An optimized register renaming mechanism is described which allows for many physical tensor registers without requiring architectural support for large tensor register names. We describe new tensor load and store instructions that reduce bandwidth requirements based on tensor dimensions. Implementations may balance data bandwidth and computation utilization for different types of tensor computations such as element-wise, depth-wise, and matrix-multiplication. We characterize the computation precision of tensor operations to balance area, generality, and accuracy loss for several well-known neural networks. The TCX processor runs at 1 GHz and sustains 8.2 Tera operations per second using a 4096 multiplication-accumulation compute unit with up to 98.83% MAC utilization. It consumes 12.8 square millimeters while dissipating 0.46 Watts per TOP in TSMC 28nm technology.**

***Index Terms*—Neural Network Accelerator, Convolutional Neural Network, ASIC Design**

I. Introduction

Deep learning is a rapidly developing branch of Artificial Intelligence (AI) that is accelerating the use of domain-specific processor architectures. Traditional processor architectures cannot keep up with the computational complexity requirements of neural networks. This has led to a proliferation of novel architectures with dramatically increased numbers of power-optimized computational units. These domain-specific architectures have improved power efficiency 10x to 100x compared with traditional implementations [1]. Furthermore, retrofitting existing Central Processing Unit (CPU) and Graphics Processing Unit (GPU) architectures for neural computations is problematic due to specialized memory systems and inconsistent programming models [2]. This has resulted in limited choices for precision and tensor operations restricting the types of neural networks that can be supported [3]. Quantization is a useful technique to reduce both the computation cost and memory budget. Eight bit integer (INT8) quantization often shows no accuracy degradation for inference and can sometimes be sufficient for training [4].

In this paper, we present the Tensor Compute Accelerator (TCX), which is a tensor computing accelerator that can be added to an existing Reduced Instruction Set Computer (RISC) CPU. It can also operate as a standalone hardware accelerator controlled by a CPU. TCX implements variable-length tensor instructions (VLT). General Processor Technologies' Unity ISA extended variable length vectors (VLV) to matrices by allowing multiple length registers [5]. TCX further extends this by allowing more than two dimensions with a datapath connected to up to 4096 compute units (CU). As with Unity, TCX implements Out-of-Order Execution (OOE) with tensor register renaming. In contrast with fixed-width SIMD, by using VLT instructions, a TCX instruction does not need to be re-fetched. Non-power-of-two length registers ensure any dimension count can be specified. A single variable length tensor instruction specifies many operations providing high utilization of Multiply-ACcumulate (MAC) units and high utilization of memory bandwidth with very low instruction bandwidth. The architecture enables both scalability of CUs and on-chip memories with software compatibility across all implementations.

This paper is organized as follows, Section II discusses related work. Section III describes the accelerator micro-architecture with integration into the Unity CPU micro-architecture. Implementation details for tensor register renaming compared to CPU register renaming are discussed. Section IV describes the programmer's view of the accelerator. Section V presents performance results and hardware efficiency. Finally, in Section VI we summarize our design and results.

II. Related Works

The high computational requirements of Deep Neural Network (DNN) applications have led to domain specific accelerators in preference to execution on general purpose processors. The Convolutional Neural Networks (CNN) are domain specific accelerators designed to improve latency, speed, bandwidth, and throughput based on specific computation patterns.

Versatile Tensor Accelerator (VTA) [6] is a hardware-software blueprint for deep learning. VTA provides a programmable template that adopts 128-bit instructions. VTA has a scalable GEMM core for matrix multiplication and a Tensor ALU for the other types of computation including activation. Compared with VTA, TCX requires a single type of computation unit reducing the number of cycles to compute convolution and activation.

VWA [7] is a CNN accelerator implemented with TSMC 40nm technology. To implement a MAC operation, it uses a 1-D broadcast dataflow followed by a three-stage accumulation. This results in extra cycles to compute stride two convolution and point-wise convolution. Additionally, VWA has dedicated fixed 4.125KB SRAM for features and dual 2.25KB for weights resulting in less efficient storage for ratios not equal to 4.5:4.125. TCX uses general purpose Direct Memory Access (DMA) operations for strided convolutions. TCX memory may contain any ratio of feature maps and weights.

III. Micro-architecture

In this section, we describe an implementation of the tensor accelerator (TCX) targeting AI inference for embedded systems. In this implementation the out-of-order Unity CPU has been extended with 64-bit TCX instructions.

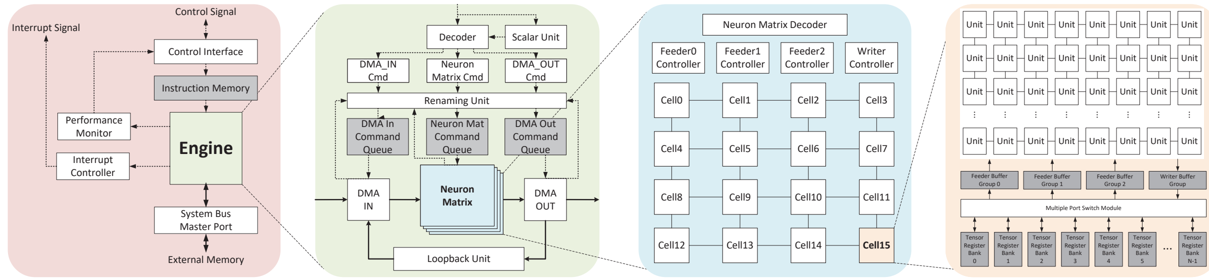

Fig. 1. TCX Micro-architecture

A. Unity CPU-TCX Partitioning

Figure 1 shows a breakout of the TCX micro-architecture and system. The left-most pink box shows the TCX engine connected to instruction memory and a system bus. Performance monitoring, interrupt, and control signals are provided. Scalar instructions and TCX instructions are fetched into a common instruction window. After instruction fetch their execution is decoupled into separate pipelines. The green box shows a scalar execution unit in the top right. It is a full 32-bit Unity out-of-order CPU. Because the TCX pipeline is decoupled from the CPU pipeline, the CPU must issue an explicit fence or barrier instruction to restore ordering of the two decoupled execute pipelines. The Unity CPU operation is not discussed further as we focus on TCX operation.

B. Out-of-Order Execution and Register Renaming

Allowing out-of-order execution of TCX instructions hides the long latency of tensor operations. When an instruction completes it is placed in a retire buffer. TCX instructions may operate on 4 dimensions of data each of which are up to 2^{64} in length. Therefore, TCX instructions are not reordered and precise exceptions are not supported.

Register renaming is widely used in out-of-order superscalar processors. Its major advantage is to resolve Write-After-Write (WAW) dependencies [8] at runtime. This is often found in programs that have no deterministic control flow or uncertain latency of operations. TCX operations, such as the loading of a tensor, can also have uncertain latency particularly if the dimension is changed at runtime.

Modern register renaming relies on Reorder Buffers (ROB) and reservation stations that can hold the forwarding values. This is impracticable for a tensor register that may be 100's of kilobytes in size. Therefore, TCX employs a variant of register renaming which eliminates the ROB and value copies in the reservation station.

When a TCX instruction is issuing, its destination architecture tensor register (ATR) will be renamed to the first physical tensor register (PTR) on the free stack. If the free stack is empty, there are no more physical registers available and issuing will be suspended. After instruction issue, the association of the 2 register IDs is updated in the mapping table. The state of the PTR is marked as "not ready". The previously associated PTR, if there is one, will be marked as in the early free state. Once the reference count of the PTR is decremented to zero, its state will be cleared and the ID will be removed from the allocated stack and pushed onto the free PTR stack. The early free state allows the allocation of a PTR as soon as there is a free one while keeping tracking of all the reference instructions that have retired. Since the number of physical registers is larger than architectural registers, architectural registers can always be bound to a physical register even if its value has been deprecated.

Unlike the scalar operands, which can directly forward the value and readiness to each pipeline stage, the tensor pipeline iterates the operation on functional units for several levels of loops which creates a small bubble (usually less than ten cycles) between the readiness signal assertion and the reading of data by the functional units. There is a similar gap between retiring an instruction and issuing a subsequent instruction.

C. Neuron Matrices

TCX organizes functional units into a physical 2-D matrix layout that is logically iterated over 3 or more dimensions. The blue box in Figure 1 shows Neuron Matrices (NR) which are the major component in TCX for executing computation instructions. The hierarchy of an NR is a 16 2-D array of cells organized into a 4×4 mesh topology, each of which operates independently. A cell in an NR is the minimum unit for logically organizing 3-D structures of function units. Each cell contains 64 compute units organized into an 8×8 mesh topology, and has complete control logic to execute independently from others. As shown in the orange box, each CU within a cell shares the same control signals. Each NR provides 1024 total CUs.

A cell mainly contains functional units in a 2-D mesh topology where units can exchange source operands with their adjacent neighbors. The dimension of each hierarchical level can be implementation defined. A 4096 compute unit implementation can be configured as 4 NRs each with 4×4 cells per NR and 8×8 CUs per cell.

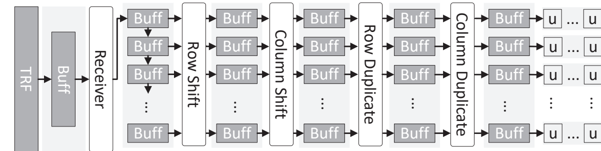

Fig. 2. TCX Buffer Pipeline

It is difficult to access a Tensor Register File (TRF) in a single cycle and to route the value to thousands of compute units. Therefore, the TRF is banked to minimize the distance between compute units and registers. This is shown in the bottom of orange box. Each bank of the TRF feeds 64 compute units in an 8×8 cell. To further reduce wire distances, the read ports are connected to functional units using 5 stage pipeline feeders as shown in Figure 2.

D. Integer Compute Unit

Figure 3 shows the implementation of the TCX Compute Unit (CU). Three operands, op0, op1, and op2, can be specified to implement $y = op0 \times op1 + op2$. TCX is primarily designed for 8-bit input features ($op0$) and kernels ($op1$), with the compatibility of mixing unsigned and signed input. 8-bit computations complete in a single cycle. This CU also supports 16-bits values as well, which take 2 cycles with the bit positions determined by the op1 high/low control lines. The internal precision of the compute unit

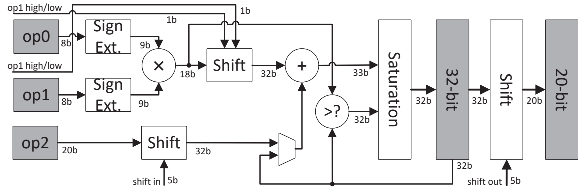

Fig. 3. Compute Unit

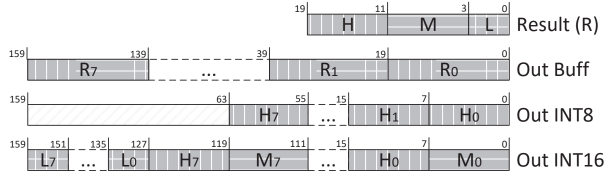

Fig. 4. CU Output Data Layout

is 32-bits. A 32-bit accumulator which may be selected in place of $op2$ is provided for multiply-accumulate operations. A 20-bit output preserves bit precision for 16-bit operands. However, when dense layers with many accumulations are present, it is possible that the full 32-bit output may be required. The shift-in signal allows the full 32-bit accumulator to be used as the op2 operand.

TCX is designed to support integer quantized computation for neural networks. The radix-point position in a fixed point representation is significant and requires shift operations to maintain the same position. The comparator in the CU enables max-pooling by treating op0 as the input and fixing op1 as 1. The maximum number is stored in the 32-bit register.

In the output stage, 20-bits from the accumulator, a 16-bit intermediate result, or an 8-bit integer may be selected. The data layout of an eight unit row is shown in Figure 4. Each 20-bit CU result consists of three parts organized into a High/Middle/Low layout. The high and middle fields are both 8-bits while the low field is 4-bits. Typically, one convolutional layer feeds in an 8-bit integer input and then generates an 8-bit activation. In this case, the data are reordered as Out INT8. All the results are sequenced in the lower 64-bits. If 16-bit integer or 20-bit accumulation is needed, the data will be organized as Out INT16.

E. Load and Store Pipeline

The tensor load/store instructions are controlled by two additional operands compared to scalar load/store instructions. These are global and local dimension operands. The global dimension operand contains the entire tensor size and can be up to 4GB. The local dimension operand contains the dimensions that can be held in a tensor register. Therefore a local dimension operand with the address of the first element defines gathering or scattering of a sub-tensor. The load/store unit is pipe-lined and distributes data between centralized TCX external ports and distributed TRF banks. Various stages of the pipeline may pad or duplicate data controlled by signals decoded from load/store instructions.

The Tensor load/store unit has several important features compared with existing architectures: 1) it reduces a large amount of address generation instructions to a single instruction, 2) implementations may combine and optimize access to memory with reduced load/store requests on the system bus, 3) VLT dimensions can be forwarded to a smart memory controller allowing combining of continuous lines and even entire matrices into a single longer transaction, and 4) it assigns an ATR ID instead of addresses for representing a sub-tensor in further instructions. This is convenient for both compiler hoisting of higher order arrays in loop optimizations and data dependency checking in out-of-order execution pipelines.

F. Loopback Pipeline

Movement between scalar registers belonging to the same register file is typically an inexpensive operation in a CPU. However, copying tensor registers can be very expensive particularly if the path requires the use of an NR. To minimize the expense, a loopback channel is used providing a direct path between the load and store pipelines. In addition to tensor register move instructions, padding, reshaping, duplication, cropping, etc. are implemented on this path.

IV. TCX Programming Model

In this section we describe how tensors are partitioned and mapped onto neural matrices, cells, and compute units. We provide different examples of configuring the cells and compute units to optimize utilization of resources. We further discuss convolutional layers with stride and pooling.

To optimize utilization we shift features between CUs and broadcast the kernel to the features. In contrast with 2-D systolic implementations [9], TCX does not use separate accumulators. Instead all compute units contain a 32-bit accumulator. CNN operations are mapped onto CUs and each CU is able to perform convolution, dot product (fully connected), pooling, and activations. Therefore, TCX does not require additional circuits for activations as in [6].

A. Partition

TCX operates on 4-D data which is decomposed into multiple 3-D sections that are executed on 2-D mesh-connected compute units. Each cell in the NR has a private switch with connections to nearest neighbors. Each NR receives data transfer signals from the decoder. The NR forms a partition of the data based on these signals such that each compute unit is mapped to a portion of the 4-D address. This allows TCX to perform a partition by receiving and executing data transfer signals from the inner decoder and thus control the dataflow in the NR. Control signals can be generated for a minimum of 8×8 sized CUs. Future implementation may increase the CU array to a maximum of 32×32.

Hardware Partition (HWP): As shown in the blue block of Figure 1, an NR is organized into a 4×4 2-D mesh of cells. The cells are physically connected to their nearest neighbors with data transfers controlled by software. Cells may be partitioned by the hardware into one of nine forms of $a \times b,\ \forall a, b \in \{1, 2, 4\}$. Each sub-partition is responsible for a block of an output feature map. Using a 1×1 partition, the 16 cells can simultaneously produce 16 feature map outputs of size 8×8. Using a 4×4 partition, the entire NR produces a single feature map output of size 32×32. The duplication bit-field $dup0$ of a load instruction duplicates data across Hardware Partitions (HWP) efficiently copying common feature maps. Inner cells of a HWP may share common data. $dup1$ loads kernels and duplicates data across cells. Partial sums typically do not require duplication. These are controlled by the $dup2$ field.

Figure 5 shows a 2×2 hardware partition denoted by the crossed lines. The 16 cells are grouped into 4 sub-partitions implementing 4 parallel output feature map (depth) computations. The output data are generated in parallel by the partitioned cells, each of which contains 16×16 CUs. A more detailed computation flow can be found in Subsection IV-B.

NR Combination: The 4096 CUs are distributed in the four homogeneous NR with respective decoders and feeders. The four neural matrices (NR) have identical hardware implementations. By

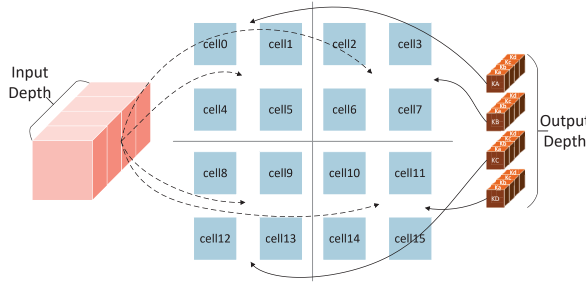

Fig. 5. Hardware Partition

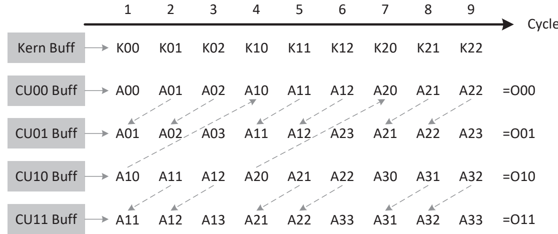

Fig. 6. Basic Convolutional Dataflow in TCX

configuring NR modes, TCX makes full use of the four NRs efficiently reducing computing latency. There are two NR combination modes - tile and expand. Tile mode shares the output depth, where kernels are equally divided into four parts along the direction of output depth, and respectively loaded into the four NRs, to generate four sets of output. Expand mode takes both the input features and kernels equally dividing them into four parts along the direction of input depth.

B. Convolution

Early CNN models deployed convolutions using a stride one sliding window resulting in a $w \times h \times c$ output feature map generated by a $k \times k \times c \times n$ kernel. This requires $w \times h \times k^2 \times c \times n$ times MAC operations.

In TCX, a five-stage buffer pipeline for data loading is used which hides the data manipulation latency. We shift the input feature and duplicate the kernel elements to implement kernel sliding instead of the input data duplication of a conventional implementation. This allows for maximum data reuse. For basic convolution mapping, input data is mapped one by one onto the CU array. Within an HWP, TCX accepts a wide range of input sizes, and theoretically, the $8N$ sized square feature realize the maximum MAC utilization, since TCX organizes the CUs by 8×8. While the kernel elements are then broadcast onto all the CUs, each element takes one cycle to finish the corresponding MAC. For example, a single feature map 3×3 kernel takes 9 cycles to finish computing while a 7×7 kernel needs 49 cycles.

Figure 6 shows a single feature map example of a basic convolution using 4 CUs (00, 01, 10, and 11). The input feature (A) is convolved by the 3×3 kernel (K) discarding padding, resulting in a 2×2 output (O). The elements are indexed by row and column. The buffers in the leftmost part of the figure represent the nearest buffers in Figure 2, where the data are shifted and duplicated by the logical circuits. In the case of a basic convolution, the kernel are broadcast into all the CUs. Each CU loads different input data and requires 9 cycles to complete the computing. The doted lines show the systolic shift between CUs. Stride two convolution (S2) changes the window stride to skip adjacent elements. This reduces computational complexity and results in less activation output data. TCX supports arbitrary strides and has specific hardware support for S2 as it is commonly used in many applications. By omitting the high 8-bits of the 16-bit input via the control of high/low selection signal, data can be read out of tensor registers with specified row and column shifts.

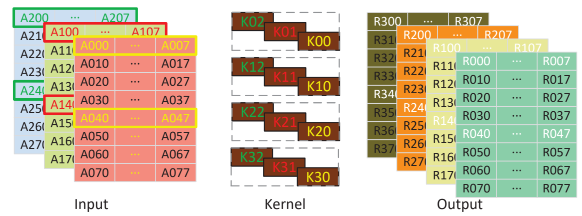

Fig. 7. Illustration of Example Point-wise Convolution

C. Fully Connected

Point-wise convolution uses a 1×1 kernel size. This can result in an imbalance when loading large feature maps. To address this issue, TCX applies a novel data fetching technique to facilitate kernel loading. Fully Connected (FC) layers are similar to 1×1 convolutional layers. A vector of neuron input data is reshaped to fit into one or more cells. This is then multiplied by a vector of weights.

Figure 7 shows a $8 \times 8 \times 3$ input feature map with $1 \times 1 \times 3 \times 4$ kernel. To illustrate TCX address operations a fourth output channel is added. Each input channel is referenced with a prefix denoting the channel $(A0, A1, A2)$ and the corresponding matrix location $(00, 01, ..., 10, 11...., \text{etc.})$. Similarly, kernel data is referenced with a prefix denoting the channel $(k0, k1, k2)$ and an index denoting which stage of convolution it is used in.

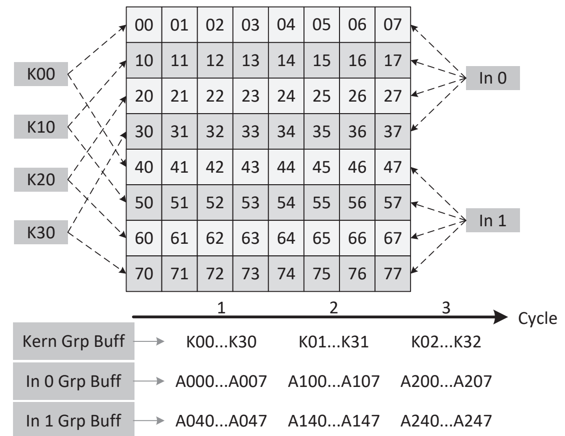

Fig. 8. Point-wise Convolutional and FCL Dataflow in TCX

Figure 8 shows the dataflow in TCX. In the first step T1, TCX loads the first input channel kernel for all the output channels (K_{n0}, $\forall n \in N$). Feature input loading is dynamically related to the input data matrix size and the number of output channels. For example, if the number of output channels N is larger than eight, only one row of the input feature data will be loaded and broadcast to all the units. In

Figure 8, $N = 4$ and one 8×8 cell has the ability to output two rows across four output channels. Therefore the first and fourth row of the first channel are loaded once and broadcast to their respective CUs. The kernel is applied to simultaneously in each CU and the result saved in the accumulator. In the second cycle T2, feature data from the second channel and kernel data operating on the first channel are loaded and broadcast. The kernel is applied and accumulated with the first channel results. Similarly the third channel is processed and accumulated. The next two rows are then loaded and the process repeats until all input feature data is processed.

D. Kernel Caching

In general, 4096 CUs will not map directly to the output size. Therefore the input tensor (both feature and kernel) has multiple partitions or divisions. Based on the kernel broadcast approach, typically the loop is in the sequence of 1) batch (multiple input images), 2) kernel divisions, and 3) feature divisions. Because the kernels are stationary, additional PTRs with tensor renaming can be used. A caching mechanism is applied to hold the kernels in CUs to reduce the bandwidth requirements of kernel loading. This reduces the kernel loading by a ratio of the number of batch size.

E. Pooling

Pooling is an operation that reduces a window of values to a single result. Pooling reduces the number of input feature values but also destroys spatial information. *Max Pooling* replaces a window of data with the maximum value contained in the window. Subsection III-D describes each compute unit. Using a comparator and shifting the input data allows the accumulator to retain the maximum value.

$$\mathbf{O}(n, w, h) = \frac{\sum_{h'=0}^{H'-1} \sum_{w'=0}^{W'-1} \mathbf{A}(c, w + w', Sh + h')}{H' \times W'} \quad (1)$$

$$= \sum_{h'=0}^{H'-1} \sum_{w'=0}^{W'-1} \left\{ \mathbf{A}(c, w + w', Sh + h') \times \frac{1}{H' \times W'} \right\} \quad (2)$$

Average Pooling reduces a window of data to the average value of the window of data. As shown in Equation 1, where $\mathbf{A}(\cdot)$ represent the tensors of input feature (activation), $\mathbf{O}(\cdot)$ the output feature, and n, c, w, h represent the indexes of output channel, input channel, output width, and height, respectively. w', h' are the width and height indexes of the feature. Their corresponding uppercase W', H' are the corresponding size, and S is the convolution stride. Average pooling is to compute the average of the entire window values. The division operation to compute the average can be transformed into multiplications. This allows normal convolution operations with an averaging kernel to be used. For a 3×3 pooling window, an element by element sum would be divided by 9. This can be transformed into a multiplication by using a 3×3 convolution kernel with the matrix values set to $1/9$.

F. Activation

Activation is a process that determines the output value by applying a function to the weighted sum of the inputs. TCX supports Linear, ReLU, Leaky ReLU, and PReLU activation functions. Linear and ReLU activation functions in TCX are implemented in the store unit of the processor because they do not require computation. Leaky ReLU and PReLU both require multiplication and are performed in compute units.

V. Evaluation

TCX is an ASIC design intended for integration in heterogeneous System on a Chip (SoC) designs. It has been fully validated on an FPGA platform and is currently being integrated into a customer SoC. This section discusses the performance and accuracy of TCX for a number of popular CNN models. Our results are obtained from evaluations on both a cycle accurate simulator (c-model) and the Synopsys HAPS emulator based on TSMC 28nm technology.

Table I shows 4 NR TCX hardware efficiency compared with several other designs. TCX power is measured at peak performance and includes both static and dynamic power. The TCX processor runs at 1 GHz and sustains 8.2 Tera operations per second. It consumes 12.8mm^2 while dissipating 0.46 Watts per TOP in TSMC 28nm technology. TCX achieves high area efficiency because of careful placement of 4096 compute units.

Table II shows the network performance of tested models on both single NR and 4 NR. The input image size for all networks are set to 256×256. Cycles gives the number of precise cycles to execute a single image on TCX. Theoretical GMACs gives the total number of multiply-accumulate operations required by the algorithm. Load gives the amount of off chip memory loaded during inference processing. Store shows the amount of data transferred into off chip memory from on chip buffers. TCX GMACs gives the overall MAC operations executed on TCX. We compute three type of utilization in the table, MAC level, cycle level, and overall utilization. We also list our total number of instructions for each inference model for reference.

Mac utilization shows the ratio of meaningful MAC operations. For example, we obtain the highest utilization of 98.84% in the 4 NR for VGG16. Memory latency and other effects such as partition mapping can result in idle CUs reducing utilization. In addition, both 1 NR and 4 NR TCX implementations generate the same size of activation because the number of CNN computations are fixed. Differences in data loading caused by convolutional operation mapping are discussed in Subsection IV-A.

VI. Conclusion

This paper presents a highly efficient programmable tensor computing accelerator - TCX. TCX accelerates neural network tensor computing - especially CNNs. A novel register renaming and computing unit design are presented using an efficient dataflow with hardware support for stride 2 and fully connected layers. TCX is scalable and programmable supporting INT8 and INT16 datatypes with extended 32-bit internal precision and 20-bit inter-CU communications. Fully quantized CNNs with up to 98.84% MAC utilization are achieved. TCX also achieved best in class power and performance compared with technology scaled recent design. The TCX processor runs at 1 GHz and sustains 8.2 Tera operations per second using a 4096 multiplication-accumulation compute units. It consumes 12.8mm^2 while dissipating 0.46 Watts per TOP in TSMC 28nm technology.

Acknowledgment

This work is partly supported by The National Key R&D Program of China (No.2018YFB0704300), Scientific and Technological Innovation Foundation of USTB (BK19AF007, BK20BF009, 06500093), NSFC (No. 61671056, No. 61971031).

References

[1] V. Sze, Y.-H. H. Chen, T.-J. J. Yang, and J. S. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 12 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8114708/

TABLE I
COMPARISON OF TCX WITH COUNTERPARTS

	TCX	VWA [7]	[10]	[11]	DNA [12]	DSIP [13]	Eyeriss [14]	[15]	[16]	[17]
Measurements	Post-layout	Post-layout	Post-layout	Post-layout	Post-layout	Chip	Chip	Chip	Chip	Chip
Technology (nm)	28	40	65	65	65	65	65	65	28	40
Precision (bits)	8 fixed	16 fixed	16 fixed	16 fixed	16 fixed	16 fixed	16 fixed	11-12 fixed	16 fixed	16 fixed
Voltage (V)	1.10	0.99	1.20	1.00	1.20	1.20	1.00	1.20	0.58	0.90
Number of CU (#)	4096	168	64	168	512	64	168	128	288	256
Clock (MHz)	1000	500	200	500	200	250	200	100	200	204
Area (mm^2)	12.80	1.56	10.60	5.00	16.00	10.56	12.25	10.52	34.10	2.40
Power (mW)	3800.0	155.0	93.4	354.0	479.0	93.4	278.0	20.5	61.0	274.0
Peak throughput (GOPS)[a]	**8192**	240	54	353	475	56	156	59	78	150
Power efficiency (GOPS/mW)[b]	**2.16**	1.25	0.69	0.82	1.18	0.72	0.46	3.45	0.35	0.37
Area efficiency (GOPS/mm^2)[c]	**640.00**	153.60	5.12	70.57	29.71	5.32	12.73	5.65	2.28	62.50

[a] Normalized by technology: result $=$ original $\times \frac{\text{technology}}{28}$ [7]

[b] Normalized by technology and voltage: result $=$ original $\times \frac{\text{technology}}{28} \times \left(\frac{\text{voltage}}{1.1}\right)^2$ [7, 18]

[c] Result by normalized throughput

TABLE II
CNN PERFORMANCE ON TCX (4NR/1NR)

Network	GoogleNet-V2	MobileNet-V1	ResNet-50	VGG16
Theo. GMACs	2.64	0.74	5.04	20.05
Cycles (M)	1.03 / 3.04	0.45 / 1.10	2.51 / 7.59	7.68 / 25.26
Insn. (K)	29.14 / 20.40	14.77 / 11.33	55.50 / 37.19	68.79 / 82.65
H/W GMACs (M)	2.83 / 2.69	0.86 / 0.84	6.57 / 6.58	20.28 / 20.28
Store (Mb)	6.32	6.28	20.32	18.79
Load (Mb)	25.64 / 24.63	11.95 / 11.03	76.08 / 66.85	207.14 / 257.10
MAC Util.	93.00% / 97.99%	86.79% / 88.16%	76.60% / 76.59%	98.84% / 98.83%
Cycle Util.	66.86% / 86.40%	46.28% / 75.13%	64.01% / 84.64%	64.50% / 78.39%
Overall Util.	62.18% / 84.66%	40.17% / 66.24%	49.03% / 64.83%	63.75% / 77.48%

[a] Mac Util. $=$ Theo. GMACs / TCX GMACs $\times$ 100%
[b] Cycle Util. $=$ TCX GMACs / (Cycle $\times$ Freq. $\times$ NR $\times$ 1024) $\times$ 100%
[c] Overall Util. $=$ Theo. GMACs / (Cycle $\times$ Freq. $\times$ NR $\times$ 1024) $\times$ 100%

[2] M. Moudgill, J. Glossner, W. Huang, C. Tian, C. Xu, N. Yang, L. Wang, T. Liang, S. Shi, X. Zhang, D. Iancu, G. Nacer, and K. Li, “Heterogeneous Edge CNN Hardware Accelerator,” in *The 12th International Conference on Wireless Communications and Signal Processing*, 2020, pp. 6–11.

[3] C.-x. Xue, T.-y. Huang, J.-s. Liu, T.-w. Chang, H.-y. Kao, J.-h. Wang, T.-w. Liu, S.-y. Wei, S.-p. Huang, W.-c. Wei, Y.-r. Chen, T.-h. Hsu, Y.-k. Chen, Y.-c. Lo, T.-h. Wen, C.-c. Lo, R.-s. Liu, C.-c. Hsieh, K.-t. Tang, and M.-f. Chang, “121-28TOPS / W for Multibit MAC Computing for Tiny AI,” *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 244–246, 2020.

[4] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, “Pruning and quantization for deep neural network acceleration: A survey,” *Neurocomputing*, vol. 461, pp. 370–403, 10 2021. [Online]. Available: http://arxiv.org/abs/2101.09671https://linkinghub.elsevier.com/retrieve/pii/S0925231221010894

[5] M. Moudgill, C. J. Glossner, A. J. Hoane, P. Hurtley, and V. Kalashnikov, “Vector processor configured to operate on variable length vectors using implicitly typed instructions,” *U.S Patent 9,959,246*, 2018.

[6] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, “A Hardware–Software Blueprint for Flexible Deep Learning Specialization,” *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 9 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8764458/

[7] K. W. Chang and T. S. Chang, “VWA: Hardware Efficient Vectorwise Accelerator for Convolutional Neural Network,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 145–154, 2020.

[8] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1 1967. [Online]. Available: http://ieeexplore.ieee.org/document/5392028/

[9] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “ShiDianNao: Shifting Vision Processing Closer to the Sensor,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, vol. 13-17-June. New York, NY, USA: ACM, 6 2015, pp. 92–104. [Online]. Available: https://dl.acm.org/doi/10.1145/2749469.2750389

[10] J. Jo, S. Kim, and I. C. Park, “Energy-Efficient Convolution Architecture Based on Rescheduled Dataflow,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4196–4207, 2018.

[11] L. Du, Y. Du, Y. Li, J. Su, Y.-C. Kuan, C.-C. Liu, and M.-c. F. Chang, “A Reconfigurable Streaming Deep Convolutional Neural Network Accelerator for Internet of Things,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 198–208, 1 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8011462/

[12] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, “Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 8 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7898402/

[13] J. Jo, S. Cha, D. Rho, and I. C. Park, “DSIP: A Scalable Inference Accelerator for Convolutional Neural Networks,” *IEEE Journal of Solid-State Circuits*, vol. 53, no. 2, pp. 605–618, 2018.

[14] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 1 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7738524/

[15] Y. Choi, D. Bae, J. Sim, S. Choi, M. Kim, L.-s. S. Kim, S. S. Member, D. Bae, J. Sim, S. S. Member, S. Choi, M. Kim, S. S. Member, L.-s. S. Kim, and S. S. Member, “Energy-Efficient Design of Processing Element for Convolutional Neural Network,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 11, pp. 1332–1336, 11 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7893765/

[16] G. Desoli, N. Chawla, T. Boesch, S.-p. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, and N. Aggarwal, “14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems,” in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2 2017, pp. 238–239. [Online]. Available: http://ieeexplore.ieee.org/document/7870349/

[17] B. Moons and M. Verhelst, “An Energy-Efficient Precision-Scalable ConvNet Processor in 40-nm CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 903–914, 4 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7801877/

[18] I. Yoo, B. Kim, and I.-C. Park, “Reverse Rate Matching for Low-Power LTE-Advanced Turbo Decoders,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 12, pp. 2920–2928, 12 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7323875/