A Dense Tensor Accelerator with Data Exchange Mesh for DNN and Vision Workloads

Yu-Sheng Lin and Wei-Chao Chen
Inventec Corporation
lin.john@inventec.com, chen.wei-chao@inventec.com

Chia-Lin Yang and Shao-Yi Chien
National Taiwan University
yangc@csie.ntu.edu.tw, sychien@media.ee.ntu.edu.tw

Abstract—We propose a dense tensor accelerator called VectorMesh, a scalable, memory-efficient architecture that can support a wide variety of DNN and computer vision workloads. Its building block is a tile execution unit (TEU), which includes dozens of processing elements (PEs) and SRAM buffers connected through a butterfly network. A mesh of FIFOs between the TEUs facilitates data exchange between tiles and promote local data to global visibility. Our design performs better according to the roofline model for CNN, GEMM, and spatial matching algorithms compared to state-of-the-art architectures. It can reduce global buffer and DRAM fetches by 2-22 times and up to 5 times, respectively.

Index Terms—Neural network hardware, vector processors, parallel programming.

I. INTRODUCTION

The goal of designing a fast deep neural network (DNN) accelerator involves packing as many processing elements (PEs) as possible and run them without stalling with a smooth and timely supply of data. While the density of PEs increases with the advancement of technology, the available DRAM bandwidth tends to grow slower than computation [1]. For effective use of this precious DRAM bandwidth, some modern accelerators exploit the sparsity in DNN to reduce DRAM bandwidth [2], [3], [4]. On the other hand, dense accelerators employ on-chip buffers that can occupy more than half of the chip area [5], [6]. Together, these resources define a performance roofline [7], and an architecture can not perform at this limit if it fetches the same data from DRAM multiple times and duplicate them in the buffers.

For this purpose, modern architectures divide a workload into individual small groups so that PE can fetch the same data repeatedly from the on-chip buffers without wasting global DRAM bandwidth. This technique is called tiling, which largely determines the characteristics of each architecture. For example, Google's Tensor Processing Unit (TPU) [5], [8] utilizes a global tiling buffer with mesh-connected PEs to create a highly scalable architecture. However, this architecture requires synchronized PE execution, resulting in bubbles when running smaller tiles in larger TPUs. A more recent TPU instead adopts a much smaller tile size [9] to alleviate this issue. Eyeriss [6], [10] is another architecture with a smaller tile size, and each PE has a dedicated local tiling buffer for computing elements of convolution partial sum (PSum) without accessing the global buffer. It employs a horizontal multicast network to deliver data to multiple local buffers within the same cycle. Because each

local buffer is private to its PE, data needs to be duplicated across several local buffers and the global buffer, wasting precious on-chip storage.

In conclusion, small tiles are good for keeping the PEs busy but bad for local buffer pressure due to data duplication. The proposed VectorMesh architecture aims to balance this tradeoff by allowing data exchange across small tiles without duplication (Fig. 1). Its basic computing block is the Tile Execution Unit (TEU), a small tile processor with synchronized PEs. Neighboring TEUs are joined together with *bidirectional FIFOs* to form a 2D mesh arrangement. VectorMesh supports classic CNN layers [11], [12], [13] and variant CNN layers [14], [15], as well as spatial matching for video inference acceleration [16], [17]. A TEU can support a tiled version of layer workloads through its two 32-bank local input buffers, 32 vectorized PEs (*PE group, PEG*), and a PSum buffer. The interconnect from the buffers to the PEG is a *butterfly network (BFN)*, and we adopt a systematical approach to guarantee BFNs can provide full throughput for our target applications. As the throughput of BFN is guaranteed, all TEUs can run at a similar speed, and each FIFO only utilizes four entries of 32 words to balance the skew. Our contributions include:

- A dense tensor accelerator that utilizes mesh FIFOs for data exchange for DNN and computer vision workloads,
- A methodology for scheduling workloads onto the proposed architecture, and
- An implementation in both SystemC and Verilog with competitive performance using the roofline analysis.

II. THE VECTORMESH ARCHITECTURE

The block diagram of VectorMesh is static and incomplete without a discussion of workload scheduling. For this, we convert the workloads into tensors (Section II-A), divide these tensors into tiles (Section II-B), share the data opportunistically using the FIFOs, and create a methodology to provide full buffer throughput to the PEs through BFNs (Section II-C).

A. Target Workloads Formulation

Matrix Multiplication (MM). Eq. (1) shows a MM operation $\mathbf{C} = \mathbf{AB}$ in a tensor form, which involves two matrices of size $O(n^2)$ and requires $O(n^3)$ operations. Each element in the 2D matrix performs a 1D dot-product, and each dot-product can be carried out in parallel. In Eq. (1), they are represented as
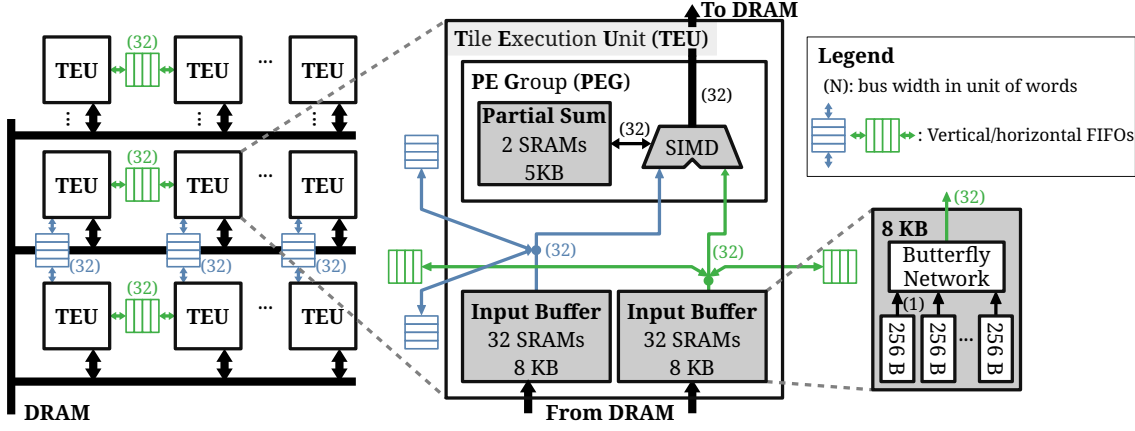
Fig. 1: **The VectorMesh architecture.** The architecture can execute the workloads such as DNN, CNN, and spatial matching. Using classic butterfly networks for data routing results in a simple yet efficient design.

two *parallel* indices (i,j) in the left-hand side expression and one *temporal* index (k) in the summation subscript.

$$\mathbf{C}(i,j) = \sum_k \mathbf{R_A}(i,j,k)\mathbf{R_B}(i,j,k),$$
where $\forall(i,j,k) \in \text{NDRange}(M,N,K),$
$$\begin{cases} \mathbf{R_A}(i,j,k) = \mathbf{A}(i,k) \\ \mathbf{R_B}(i,j,k) = \mathbf{B}(k,j) \end{cases}. \quad (1)$$

Convolution Neural Network (CNN). A CNN layer produces an output feature tensor $\mathbf{C}$ ($C_o \times o_w \times o_h$) using a kernel tensor $\mathbf{k}$ ($C_o \times C_i \times k_w \times k_h$) and an input feature tensor $\mathbf{I}$ ($C_i \times i_h \times i_w$). Each output element requires a $C_i \times k_w \times k_h$ convolution, which can also be written in a similar form:

$$\mathbf{C}(i,j,k) = \sum_{l,m,n} \mathbf{R_I}(i,j,k,l,m,n)\mathbf{R_k}(i,j,k,l,m,n),$$
where $\forall(i,j,k,l,m,n) \in \text{NDRange}(C_o,o_w,o_h,C_i,k_w,k_h),$
$$\begin{cases} \mathbf{R_I}(i,j,k,l,m,n) = \mathbf{I}(l,j+m,k+n) \\ \mathbf{R_k}(i,j,k,l,m,n) = \mathbf{k}(i,l,m,n) \end{cases}. \quad (2)$$

Spatial Matching Algorithms. Special matching algorithms [16], [17], [18] have huge potentials when combined with modern CNNs. These algorithms require two input feature maps, namely *current* and *reference*. For each pixel (block) in the *current* feature map, we compute the dot-product between the *current* pixel and its nearby pixels in the *reference* feature map. For example, the correlation layer [16] computes the spatial correlation between two tensors $\mathbf{I}$ ($C_i \times o_h \times o_w$), which we can write as:

$$\mathbf{C}(i,j,k,l) = \sum_m \mathbf{R_{I1}}(i,j,k,l,m)\mathbf{R_{I2}}(i,j,k,l,m),$$
where $\forall(i,j,k,l,m) \in \text{NDRange}(s_w,s_h,o_w,o_h,C_i),$
$$\begin{cases} \mathbf{R_{I1}}(i,j,k,l,m) = \mathbf{I}_1(m,i,j) \\ \mathbf{R_{I2}}(i,j,k,l,m) = \mathbf{I}_2(m,i+k,j+l) \end{cases}. \quad (3)$$

Spatial matching algorithms require different datapath designs and therefore cannot be efficiently supported by CNN or MM processors [19], [20].

B. Tiled Execution for Target Workloads

To make VectorMesh adaptive to these target workloads, it must allow various permutations and combinations of the *parallel* and *temporal* indices in any order. This section illustrates a methodology to convert the mathematical forms above into a feasible workload scheduling in VectorMesh.

A VectorMesh TEU has 16 KB input buffers and a 5 KB PSum buffer available for tiling. To obtain a valid tiling scheme, we must divide workloads into groups that fit into the buffers. Take the MM workload (Eq. (1)) as an example. A natural tiling is to divide the NDRange into rectangular groups of size (t_i,t_j,t_k):

$$\mathbf{C}(i,j) = \sum_k \mathbf{R_A}(i,j,k)\mathbf{R_B}(i,j,k),$$
where $\forall(i,j,k) \in \text{NDRange}(t_i,t_j,t_k). \quad (4)$

Based on tensor analysis methodologies [21], [22], this results in $t_i t_j t_k$ MAC operations on $t_i t_j$ PSums and $(t_i + t_j)t_k$ input buffers. We keep PSums with the same *parallel* indices (i,j) static in a TEU, such that the PSum does not consumes external bandwidth, and one MAC operation consumes $(t_i + t_j)t_k/(t_i t_j t_k)$ bandwidth on average. This methodology also applies to other target workloads, and we can manually choose a valid tile size that minimizes the bandwidth for every target workloads.

This scheduling results in only one external memory write for each PSum, which is the optimal bandwidth. On the other hand, the input buffer size limits the available tile size. To overcome the limitation, we add the FIFO to share the input buffer across TEUs.

Consider executing this GEMM example on VectorMesh:

$$\underbrace{\begin{bmatrix} \mathbf{P} & \mathbf{Q} \\ \mathbf{R} & \mathbf{S} \end{bmatrix}}_{\mathbf{C}} = \underbrace{\begin{bmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \mathbf{W} & \mathbf{X} \\ \mathbf{Y} & \mathbf{Z} \end{bmatrix}}_{\mathbf{B}}. \quad (5)$$

Fig. 2 shows a possible scheduling for sharing the input buffer. We need four TEUs to compute the output $(\mathbf{P},\mathbf{Q},\mathbf{R},\mathbf{S})$. In Fig. 2a, the PSums $\mathbf{P} = \mathbf{EW}$ and $\mathbf{Q} = \mathbf{EX}$ would both request $\mathbf{E}$, which can be shared through the horizontal FIFOs under

this scheduling. To identify all shareable tensors, we first notice that the upper two TEUs process tiles with different *parallel* indices j. Since *the partial derivative of the right-hand side arguments of* **A** *against j is zero in* Eq. (1) (*i.e.*, $\partial(i,k)/\partial j = \mathbf{0}$), the upper two TEUs only need to read **E** once.

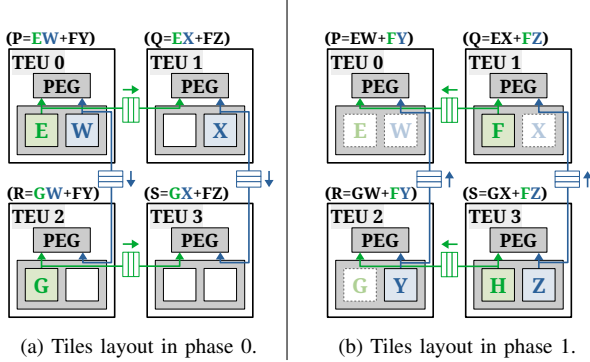(a) Tiles layout in phase 0.   (b) Tiles layout in phase 1.

Fig. 2: The data sharing mechanism ensure all input buffer store unique data, maximizing the utilization of SRAMs.

C. Executing a Tile on a TEU

A TEU has 32 vectorized PEs and can consume 32 *parallel* indices from a tile per cycle. Using the same MM representation, we have:

$$\mathbf{C}(i,j) = \mathbf{C}(i,j) + \mathbf{R_A}(i,j,k)\mathbf{R_B}(i,j,k), \tag{6}$$
$$\forall(i,j) \in \mathrm{NDRange}(p_i, p_j), \quad p_i p_j = 32.$$

To perform this operation, PEs must read two vectors from the input buffers through the BFN [23], [24]. However, if some TEUs cannot access its vectors in one cycle, then stall occurs and propagates to all TEUs. To prevent this from happening, we adopt a strategy as follows. For a memory system consisting of 2^X banked SRAM ($X = 5$ in VectorMesh), Lin *et al.* [23] shows that if the accessing address A_N of the N-th PE can be written as $A_N = A_0 + \sum_{i=0}^{X} 2^X o_i b_i$, where b_i is the i-th digits of the binary notation of N, and o_i's are odd numbers, then a BFN can always serve the data for this system in one cycle. Applying the padding and shuffling techniques to the local buffer data [23], [25], we ensure all our target workloads can fulfill this condition.

III. EXPERIMENTS

A. Workloads Supported by VectorMesh

VectorMesh supports a broader range of workloads compared with classic CNN accelerators like TPU or Eyeriss. In this section, we first benchmark the performance for *typical DNN workloads* across different architectures. Next, we show that VectorMesh can achieve high performance for *modern CNN workloads and spatial matching workloads* that do not execute efficiently on other architectures. The following paragraphs describe these two types of workloads.

Typical DNN Workloads. We select representative DNN layers from AlexNet, TinyYOLO, Inception, and SRCNN [11],

[12], [13], [26]. (We abbreviate them to AL, TY, IN, and SR, respectively.) As shown in Table I, these workloads cover both square and non-square kernels with sizes $(1, 3 \cdots, 11)$.

TABLE I: Classic CNN workloads for benchmarking.

Layer	Stride s	Kernel k_w, k_h	Channel C_i, C_o	Layer	Stride s	Kernel k_w, k_h	Channel C_i, C_o
AL CONV1	4	11,11	3,48	TY CONV4	1	3,3	64,128
AL CONV2	1	5,5	48,128	TY CONV5	1	3,3	128,256
AL CONV3	1	3,3	128,192	TY CONV6	1	3,3	256,512
AL CONV4	1	3,3	192,192	TY CONV8	1	1,1	1024,125
AL CONV5	1	3,3	192,128	IN 1×7	1	1,7	64,64
TY CONV1	1	3,3	3,16	IN 7×1	1	7,1	64,64
TY CONV2	1	3,3	16,32	SR CONV1	1	9,9	3,64
TY CONV3	1	3,3	32,64				

Modern CNN Workloads and Spatial Matching Workloads. We select the representative DNN layers from more recent networks including the DeepLab [27], ESPCN [14], and MobileNet [28]. Also, we select spatial matching workloads [16], [17] discussed in Section II-A.

B. Architectural Implementation

We develop a cycle-level simulator to evaluate the effectiveness of the proposed VectorMesh architecture. We also implementation efficient TPU and Eyeriss simulator by tiling and prefetching. For example, the normalized performance of AlexNet on our 128-PE Eyeriss only differs slightly (10%) from the reference implementation [6].

The detailed simulation configuration is explained below:

- **PE Numbers and Frequency.** We simulate $N_{PE} = 128$ and 512 PEs version for TPU, Eyeriss, and VectorMesh. TPU and Eyeriss are shaped as 8×16, 16×32 PEs, and VectorMesh is shaped as 2×2 and 4×4 TEUs. The simulation also assumes a working frequency of 200 MHz.

- **Bandwidth and Buffer Sizes.** We adopt a DDR simulator [29] to more accurately evaluate the DRAM and global buffer subsystem. We adopts fixed 6.4GB/s DRAM bandwidth (two DDR4-1600 x16 devices) and 25.6 GB/s global buffer bandwidth. For every PE in TPU, Eyeriss, and VectorMesh, we allocate $0, 0.3, 0.6$ KB local buffers and $1.0 N_{PE}, 0.5 N_{PE}, 2$ KB global buffers, respectively. We choose these numbers to match the PE-to-memory ratio from existing publications [5], [6]. Also, the required size of global buffer of VectorMesh does not need to grow with N_{PE} accordingly.

For the circuit-level comparison, we implement VectorMesh with Verilog. As a result of the architecture's simplicity, our codes are relatively lightweight at 9.6k lines, using 1.1M gates per TEU. Based on our synthesizing results and the statistics from Eyeriss and TPU, we also estimate the area of different architecture given the buffer configuration above, as shown in Table II.

TABLE II: Area estimation of the architectures.

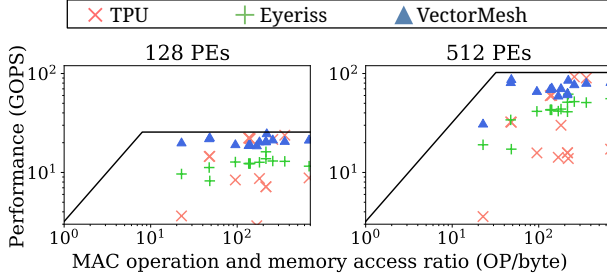|	Eyeriss	TPU	VectorMesh
MAC	0.08	0.08	0.08
Global buffer	0.19	0.38	0.00
Local buffer	0.48	0.00	0.67
Controllers	0.25	0.00	0.25
BFN+FIFO	0.00	0.00	0.04
Area factor (A)	1.00	0.46	1.04

Fig. 3: **Architectural comparisons using the roofline analysis against the workloads in Table I.**

C. Architectural Simulation

Based on the simulator and the hardware implementation, we discuss how different architectures can fully utilize their computation resources efficiency from three aspects:

Roofline Analysis. The roofline analysis provides an upper-bound under the same bandwidth and PE resources for different workloads. Given an infinite size and infinite bandwidth on-chip buffer, the performance upper-bound of a workload is the minimum between (1) the PE processing rate over the total MAC operations and (2) the DRAM bandwidth over total input and output data sizes. In Fig. 3, we denote the roofline as the black line. As can be seen, VectorMesh performs closer to the roofline than others when given the same resources because it is more bandwidth efficient, as discussed in the next paragraph.

Memory Access. Lower memory access indicates higher utilization and lower power. We define the *normalized access* as the number of bytes accessed from memory per 1,000 MAC operations. We calculate this for both global buffers (GLB) and the DRAM. As shown in Table III, in terms of global buffer bandwidth, TPU generates 18-22x larger bandwidth than VectorMesh due to the lack of local buffer; VectorMesh consumes 2-4x less bandwidth than Eyeriss since it does not duplicate data in local buffers. As a result, even with 64-256x smaller global buffers than TPU and Eyeriss, our DRAM bandwidth result is still competitive, with -14-$+44\%$ and 2-5x bandwidth reduction compared with Eyeriss and TPU, respectively.

Area Efficiency. Since different architectures require different hardware resources per PEs, for a fair comparison, it is crucial to take area efficiency into account as it reflects how much
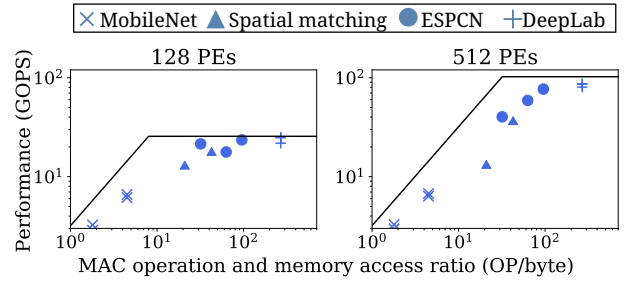

Fig. 4: **Roofline analysis for workloads supported exclusively by VectorMesh.** The performance results of VectorMesh can reach the roofline for both memory-bounded and computation-bounded workloads.

performance is available per chip area. We define this metric by dividing its average performance P on all workloads by an area factor A based on the estimation in Table II. As shown in Table III, while Eyeriss and VectorMesh provide better raw performance, only VectorMesh provides higher area efficiency that TPU since Eyeriss suffers from data duplication across its local buffers. Also, a 128-PE TPU provides a higher area efficiency than a 128-PE VectorMesh because when the number of PEs is small, both architectures utilize small tiles, which match the workload. The routing simplicity of TPU thus produces a marginally better efficiency.

TABLE III: The overall comparison of DNN architectures.

	128 PE			512 PE		
	TPU	Eyeriss	Vector-Mesh	TPU	Eyeriss	Vector-Mesh
Normalized GLB access	935	160	**42**	534	55	**29**
Normalized DRAM access	239	85	**45**	71	**28**	32
Area-efficiency (P/AN)	**22.55**	12.48	20.49	15.91	11.12	**17.31**
Performance (P, GOPS)	10	12	**20**	27	41	**68**
Area factor (A)	0.46	1.00	1.04	0.46	1.00	1.04
Area multiplier (N)	1	1	1	4	4	4

D. Workloads Adaptiveness of VectorMesh

In Fig. 4, the roofline analysis also shows that VectorMesh can smoothly process layers with highly computation-bounded in recent networks like ESPCN [14] and DeepLab [27]. For layers in MobileNet [28], while the performance results are relatively low, we have already reached the roofline. This figure also demonstrates the spatial matching workloads at reasonably optimal performance.

IV. CONCLUSION

In this paper, we proposed the VectorMesh architecture, discussed a workload scheduling process, and demonstrated its ability to execute various DNN and vision workloads closer to the performance roofline than other state-of-the-art architectures. We provided practical implementations of the architecture and demonstrate its ability to reduce global buffer and DRAM fetches by 2-22 times and up to 5 times, respectively.

References

[1] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). volume 57, pages 10–14, 02 2014.

[2] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. pages 27–40, 06 2017.

[3] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, June 2016.

[4] Y. S. Lin, H. C. Lu, Y. B. Tsao, Y. M. Chih, W. C. Chen, and S. Y. Chien. Gratetile: Efficient sparse tensor tiling for cnn processing. In *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, 2020.

[5] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12, 2017.

[6] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[7] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[8] S. Y. Kung. *VLSI array processors*. 1988.

[9] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7):67–78, June 2020.

[10] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks. *CoRR*, abs/1807.07928, 2018.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[12] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.

[13] C. Dong, C. C. Loy, K. He, and X. Tang. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307, Feb 2016.

[14] Wenzhe Shi, Jose Caballero, Ferenc Huszar, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[15] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *ICLR*, 2016.

[16] A. Dosovitskiy, P. Fischer, E. Ilg, P. Häusser, C. Hazırbaş, V. Golkov, P. v.d. Smagt, D. Cremers, and T. Brox. FlowNet: Learning optical flow with convolutional networks. In *IEEE International Conference on Computer Vision (ICCV)*, 2015.

[17] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. EVA^2 : Exploiting temporal redundancy in live computer vision. *CoRR*, abs/1803.06312, 2018.

[18] C. Rhemann, A. Hosni, M. Bleyer, C. Rother, and M. Gelautz. Fast cost-volume filtering for visual correspondence and beyond. CVPR '11, pages 3017–3024, Washington, DC, USA, 2011. IEEE Computer Society.

[19] T. Komarek and P. Pirsch. Array architectures for block matching algorithms. *IEEE Transactions on Circuits and Systems*, 36(10):1301–1308, 1989.

[20] L. De Vos and M. Schobinger. VLSI architecture for a flexible block matching processor. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(5):417–428, 1995.

[21] Y. S. Lin, W. C. Chen, and S. Y. Chien. Unrolled memory inner-products: An abstract gpu operator for efficient vision-related computations. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 4587–4595, Oct 2017.

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

[23] Y. Lin, W. Chen, and S. Chien. MERIT: Tensor transform for memory-efficient vision processing on parallel architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–14, 2019.

[24] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 461–475, New York, NY, USA, 2018. ACM.

[25] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[26] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *AAAI Conference on Artificial Intelligence*, 02 2016.

[27] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. 06 2017.

[28] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[29] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, Jan 2016.