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 TPs. A more recent TPU instead adopts a much smaller tile size [9] to alleviate this issue. Eyervisor [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 \( C = AB \) 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...
two parallel indices \((i, j)\) in the left-hand side expression and one temporal index \((k)\) in the summation subscript.

\[
C(i, j) = \sum_{k} R_A(i, j, k) R_B(i, j, k),
\]

where \(\forall (i, j, k) \in \text{NDRange}(M, N, K)\),

\[
\begin{cases} 
R_A(i, j, k) = A(i, k) \quad \text{and} \\ 
R_B(i, j, k) = B(k, j).
\end{cases}
\]

### Convolutional Neural Network (CNN)

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

\[
C(i, j, k) = \sum_{l,m,n} R_l(i, j, k, l, m, n) R_k(i, j, k, l, m, n),
\]

where \(\forall (i, j, k, l, m, n) \in \text{NDRange}(C_o \times o_w \times o_h, C_i \times k_w \times k_h)\),

\[
\begin{cases} 
R_l(i, j, k, l, m, n) = I_l(i, j + m, k + n) \quad \text{and} \\ 
R_k(i, j, k, l, m, n) = K(l, i, l, m, n).
\end{cases}
\]

### Spatial Matching Algorithms

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 \(I(C_i \times o_w \times o_h)\), which we can write as:

\[
C(i, j, k) = \sum_{l,m} R_{11}(i, j, k, l, m) R_{12}(i, j, k, l, m),
\]

where \(\forall (i, j, k, l, m) \in \text{NDRange}(s_w \times s_h \times o_w, o_h, C_i)\),

\[
\begin{cases} 
R_{11}(i, j, k, l, m) = I_1(m, i, j) \quad \text{and} \\ 
R_{12}(i, j, k, l, m) = I_2(m, i + k, j + l).
\end{cases}
\]

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

### 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 \((t_i, t_j, t_k)\):

\[
C(i, j) = \sum_{k} R_A(i, j, k) R_B(i, j, k),
\]

where \(\forall (i, j, k) \in \text{NDRange}(t_i, t_j, t_k)\).

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 consume 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 workload.

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:

\[
\begin{bmatrix}
P & Q \\
R & S
\end{bmatrix}
= 
\begin{bmatrix}
E & F \\
W & X
\end{bmatrix}
\begin{bmatrix}
A \\
B
\end{bmatrix}.
\]

Fig. 2 shows a possible scheduling for sharing the input buffer. We need four TEUs to compute the output \((P, Q, R, S)\). In Fig. 2a, the PSums \(P = EW\) and \(Q = EX\) would both request \(E\), 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., \( \partial(i,k)/\partial j = 0 \)), the upper two TEUs only need to read \( E \) once.

![Tiles layout in phase 0.](image1)

(a) Tiles layout in phase 0.

![Tiles layout in phase 1.](image2)

(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:

\[
C(i,j) = C(i,j) + A(i,j,k)R(k,i,j), \quad \forall (i,j) \in \text{NDRange}(p_i,p_j), \quad p_i,p_j = 32.
\]

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 \( A_N = A_0 + \sum_{i=0}^{N-1} 2^i r_i b_i \), where \( b_i \) is the \( i \)-th digits of the binary notation of \( N \), and \( r_i \)’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 \((1,3\cdots,11)\).

| Layer | Stride | Kernel | Channel |
|-------|--------|--------|---------|
| AL CONV1 | 1 | 5,5 | 48,128 |
| AL CONV2 | 1 | 3,3 | 128,192 |
| AL CONV3 | 1 | 3,3 | 92,192 |
| AL CONV4 | 1 | 3,3 | 92,192 |
| AL CONV5 | 1 | 3,3 | 128,192 |
| TY CONV1 | 1 | 3,3 | 3,16 |
| TY CONV2 | 1 | 3,3 | 16,32 |
| TY CONV3 | 1 | 3,3 | 32,64 |
| SR CONV1 | 1 | 9,9 | 3,64 |

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 \( 8 \times 16 \), \( 16 \times 32 \) PEs, and VectorMesh is shaped as \( 2 \times 2 \) and \( 4 \times 4 \) 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, 0.3, 0.6 KB local buffers and 1.0N_{PE}, 0.5N_{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 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 | VectorMesh | TPU | Eyeriss | VectorMesh |
| 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.
