Morphling: A Reconfigurable Architecture for Tensor Computation

Liqiang Lu and Yun Liang, *Senior Member, IEEE*

Abstract*—Tensor algebra plays a major role in various applications, including data analysis, machine learning, and hydrodynamics simulation. Different tensor algebra inherently varies in dimension, size, and computation, leading to different execution preference, including parallelization, data arrangement, and accumulation. Another critical aspect for tensor algebra is the involved tensors can be with varying mixes of dense and sparse representation. Such diversified applications are notoriously difficult to accelerate. Prior ASIC architectures do not meet the needs due to fixed dataflow and prior fine-grained fabrics (e.g., FPGAs) solutions offer limited performance and power improvement due to bit-level reconfigurable structure. In this article, we propose Morphling, a reconfigurable architecture that can flexibly handle both dense and sparse tensor computation. We first generalize a flexible execution model that decomposes tensor operations into three steps, including tensor vectorization, vector computation, and output reduction. The dense and sparse tensor computation share the same execution model, but differ in the vector computation step where the multiplications are conducted. Depending on the number of inputs and outputs that are linked together in the computation step, we define three parallel patterns, including many-to-one, one-to-many, and one-to-one, which correspond to different implementations for dense and sparse computation. Furthermore, to efficiently support sparse tensor, we design a tiled-BCSR format that enables high parallelism and balanced workload. At the architecture level, we propose a reconfigurable design to support the execution model. The hardware units can be reconfigured to support different datapath and enable different types of data reuse. We evaluate Morphling using various tensor operations and compare it with CPU, GPU, FPGA, and state-of-the-art ASIC designs. Overall, Morphling achieves 13.4X, 677.7X, 44.7X energy efficiency over Xilinx ZC706 FPGA, Intel i7-9700K CPU, and NVIDIA TitanX GPU.

***Index Terms*—Accelerator architectures, data flow computing, reconfigurable architectures, sparse matrices.**

I. Introduction

Tensor algebra is a powerful tool in many applications, such as data analysis, machine learning, and hydrodynamics simulation [1], [2], [19], [33], [37], [54], [71], [79], [84]. Distinct tensor algebra exhibits inherent variation in tensor dimension, computation, and accumulation. For example, 2D-convolution (2D-CONV) and general-purpose matrix multiplication (GEMM) are two frequently used kernels in modern complex DNNs, such as Resnet [31], GoogLeNet [76], and data analysis [6], [56]. 2D-CONV in Resnet involves 4-order tensors and 3-order tensors with sizes ranging from 10 to 1K [31], while GEMM contains three 2-order matrices and the size of the matrices can be much larger, e.g., Filter3D dataset involves 106K $\times$ 106K matrices [56]. In 2D-CONV, the 2D-kernel slides through the feature map where the elements inside the sliding window conduct a multiply-and-accumulation (MAC) operation to generate a single output, while in GEMM each output element is generated by accumulating the multiplication results from one row of a matrix and one column of another matrix. Another important feature for tensor algebra is the tensors can be with different mixes of dense and sparse representation. For sparse computation, the sparsity can vary hugely for different tensors. For example, for deep learning algorithms, the sparsity of 2D-CONV can vary from 30% to 95% for input tensor depending on the used pruning techniques [27], [28]; for tensor factorization algorithms, the tensors exhibit different degrees of sparsity for different dataset [6], [56].

Variation in tensor algebra leads to different execution preference, including parallelization, data arrangement, and accumulation. We compare the hardware efficiency of two widely used parallelization strategies for 2D-CONV and GEMM in Fig. 1. Dot-product (DP) parallelization yields a single value by multiplying and accumulating elements from two vectors, which is widely used in prior accelerator designs [25], [26], [35], [42], [43], [48], [49], [52], [80]–[82], [85], [89]. Outer-product (OP) parallelization returns a matrix where each element in one vector is multiplied with all the elements in the other vector [53], [61], [62]. Both two parallelization strategies can be used for executing 2D-CONV and GEMM by transforming the tensors as shown in Fig. 2. Fig. 1(a) computes the hardware efficiency of these two parallelization strategies for the first 24 layers of GoogLeNet [76] using 2D-CONV. The hardware efficiency refers to the utilization of MAC units. We observe that no single parallelization strategy wins all cases. In particular, OP wins for 18 layers while DP wins for six layers. For GEMM, the x-axis in Fig. 1(b) represents different shapes of the matrices including regular, tall-skin, and short-fat matrices. When multiplying matrix A ($M \times K$) with matrix B ($K \times N$), DP parallelization achieves high efficiency when K is large while OP is better when M and N are large. The difference in hardware efficiency for different parallelization strategies is caused by the variation in the size of different tensors.

Manuscript received 30 July 2021; revised 30 September 2021; accepted 7 December 2021. Date of publication 14 December 2021; date of current version 24 October 2022. This work was supported in part by the Beijing Natural Science Foundation under Grant JQ19014, and in part by the PKU-Baidu Fund under Project 2020BD024. This article was recommended by Associate Editor P. Brisk. *(Corresponding author: Yun Liang.)*

The authors are with the Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, Beijing 100871, China (e-mail: liqianglu@pku.edu.cn; ericlyun@pku.edu.cn).

Digital Object Identifier 10.1109/TCAD.2021.3135322

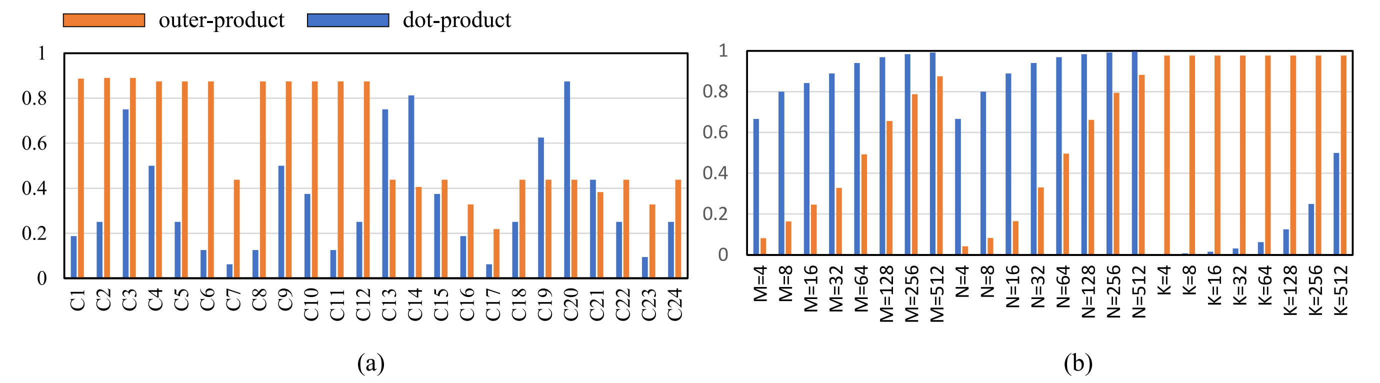


Fig. 1. Hardware efficiency for 2D-CONV and GEMM. C1-C24 are the first 24 layers of GoogLeNet [76]. For GEMM, when one dimension changes, the other two are set to 4096. The architecture is assumed to have 512 MAC units. For OP parallelization, we apply output stationary systolic array architecture. For DP parallelization, we apply input stationary systolic array architecture. (a) 2D-CONV tensor operation. (b) GEMM tensor operation.

Meanwhile, it is well established that tensor algebra involves overwhelming computation [6], [22], [56]. Traditional accelerators, such as CPUs and GPUs have been employed to accelerate tensor operations which suffer from low energy efficiency [10], [64]. Dedicated ASIC accelerators solve this problem, but lose flexibility to handle various tensor applications [11], [14], [18], [34], [48]. The requirement for flexibility and efficiency motivates the idea of accelerating tensor algebra using reconfigurable architectures. FPGAs are reconfigurable architectures that provide bit-level reconfigurability in logic blocks. However, this fine-grained reconfigurability results in a high area and power overheads [9], [38], [39], [65]. To solve these architectural inefficiencies, coarse-grain reconfigurable architectures (CGRAs) realize the best possible tradeoff between flexibility and efficiency, which use word-level reconfigurability and contain larger logic blocks and datapath-oriented interconnections.

To accelerate diverse tensor algebra on hardware, we introduce a tensor-specific CGRA framework, which can accelerate various tensor operations with arbitrary dimension, size, and sparsity. We first propose a flexible tensor execution model that generalizes the tensor operation into three steps, including vectorization, computation, and reduction for both dense and sparse tensor computation. The vectorization step rearranges the original tensor into vectors for parallelization. The reduction step accumulates the outputs in the computation step and generates the final results. Depending on the number of inputs and outputs that are linked together in the computation step, we abstract three parallel patterns, including many-to-one, one-to-many, and one-to-one. The three parallel patterns have the flexibility to choose different implementations for dense and sparse tensor computation. For dense operations, they correspond to DP, OP, and element-wise vector multiplication (EWVM), respectively. For sparse operations, they correspond to the row-wise product, Kronecker product [75], and block-wise multiplication, respectively. To efficiently support sparsity, we also propose a tiled-Block Compress Sparse Row (tiled-BCSR) format, where the nonzeros are first packed in blocks and then organized in tiles. This format leads to regular accumulation and data access patterns. Besides, blocks are evenly distributed in tiles, which helps to balance the workload and provide high parallelism.

Traditional vectorization is a linear transformation which converts a matrix into a column vector. In this work, our vectorization step duplicates tensor elements with a certain manner to unify the tensor computation pattern. This duplication also enables data reuse during the computation. Though many spatial architectures support vector operators like DP, OP, they did not answer how to decompose tensor applications into these vector operators and gather their partial sums. Our contribution is to provide an execution model integrated with a reconfigurable architecture that can formulate different dataflow for a wide range of tensor applications.

At the architecture level, we propose a CGRA design with a reconfigurable PE array and a reconfigurable network for data communication to implement the execution model. Each PE is responsible for the vectorization and computation step, while the reduction step is implemented as the nPE data communication. The PE features a reconfigurable adder tree to gather the partial sum in different manners by controlling the forward data of each adder. The communication network is a 2-D array of switches, which is used for the reduction step and data communication among PEs. Each switch contains local buffers and accumulator lanes (ALs). The local buffer can store either the input data or the results of adjacent PEs, which provides multidimensional data reuse. The ALs can be cooperated to conduct different accumulation patterns. By configuring the PE array and communication network, Morphling can support a wide range of hardware dataflow represented in our execution model. Finally we apply polyhedral model for application mapping, which takes tensor notation as the input and explores different loop transformations under architectural constraints. Prior CGRA designs are mainly designed for general applications [17], [23], [29], [30], [83]. Morphling is a domain-specific CGRA architecture with specialization in computation and accumulation in the PE and communication design and flexibility in the execution model and compiler mapping for tensor computation.

In summary, this article makes the following contributions.

1) We propose a flexible domain-specific CGRA architecture and compiler mapping for sparse and dense

TABLE I
TENSOR SIZE, COMPUTATION OF COMMON TENSOR OPERATIONS. THE COLORED DIMENSIONS INVOLVE INDEX GATHER OPERATORS. STEP 1: VECTORIZATION STEP. STEP 2: COMPUTATION STEP. STEP 3: REDUCTION STEP

Tensor operation	**Tensor size**	**Computation**	**Execution model**		
			STEP 1	**STEP 2**	**STEP 3**
GEMM$_1$	Y: N×M; A: N×K; B: K×M	$Y(i,j)=\sum_k A(i,k)*B(k,j)$	A(i,:),B(:,j)	DP	NA
GEMM$_2$	Y: N×M; A: N×K; B: K×M	$Y(i,j)=\sum_k A(i,k)*B(k,j)$	A(:,k),B(k,:)	OP	FA
2D-CONV$_1$	Y: N×H×W A: N×M×P×Q; B: M×H×W	$Y(i,j,n)=\sum_m\sum_p\sum_q A(n,m,p,q)*B(m,i+p,j+q)$	A(:,:,p,q), B(:,i+p,j+q)	DP	PA
2D-CONV$_2$	Y: N×H×W A: N×M×P×Q; B: M×H×W	$Y(i,j,n)=\sum_m\sum_p\sum_q A(n,m,p,q)*B(m,i+p,j+q)$	A(:,m,:,:), B(m,:,:)	OP	PA
KRP	Y: NP×MQ; A: N×M; B: P×Q	$Y(i*P+x,j*Q+y)=A(i,j)*B(x,y)$	A(i,:),B(i,:)	OP	NA
3D-CONV	Y: H×W×K A: P×Q×R; B: H×W×K	$Y(i,j,k)=\sum_p\sum_q\sum_r A(p,q,r)*B(i+p,j+q,k+r)$	A(p,q,:), B(i+p,j+q,:)	DP	PA
SpMM	Y: N×M; A: N×K; B: K×M A, B in CSR	$Y(i,B_{idx}(k))=\sum_{j=A_{ptr}(i)}^{A_{ptr}(i+1)}\sum_{k=B_{ptr}[A_{idx}(j)]}^{B_{ptr}(A_{idx}(j)+1)} A_{val}(j)*B_{val}(k)$	$A_{ptr}(:)$	-	NA
Stencil-Jacobs	Y: N×M; A: N×M	$Y(i,j)=A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))$	A(:,:)	EW	NA
EWMM	Y: N×M; A: N×M; B: N×M	$Y(i,j)=A(i,j)*B(i,j)$	A(i,:),B(i,:)	EW	NA

MTTKRP	Y: N×M; A: N×P×Q
	B: Q×M; C: P×M;
		$Y(i,j)=\sum_p\sum_q A(i,p,q)*B(p,j)*C(q,j)$	A(i,p,:),C(:,j)
			A(i,:,q),B(:,j)
				DP	NA

GEMM: general purpose matrix multiplication, used in deep learning, data analysis.
2D/3D-CONV: two/three-dimensional convolution, used in deep learning, image processing.
KRP: Khatri-Rao product, used in tensor fabrication.
SpMM: sparse-sparse matrix multiplication, used in data base, deep learning.
MTTKRP: matricized tensor times Khatri-Rao product, used in recommendation systems, dimensionality reduction.
EWMM: element-wise matrix multiplication.

tensor applications. Morphling can handle different tensor algebra with variations in dimension, computation, and representation.

2) We define a flexible execution model to generalize the tensor algebra to a programmable form and design a reconfigurable architecture to support this.
3) We propose tile-BCSR format where the data in the block shows regular accumulation and access patterns. Tiled-BCSR first packs the nonzeros into blocks and then stores in tiles.

We evaluate Morphling using various tensor operations and compare it with CPU, GPU, FPGA, and state-of-the-art ASIC design. Overall, Morphling achieves 13.4X, 677.7X, 44.7X energy efficiency over Xilinx ZC706 FPGA, Intel i7-9700K CPU, NVIDIA TitanX GPU.

II. BACKGROUND

Tensor is defined as matrices to any number of dimensions. The number of dimensions is defined as its order. For example, a scalar is a zero-order tensor and a vector is a one-order tensor. Table I lists eight widely used tensor operations in two categories. The first six operations require partial sum accumulations while the last two only involve multiplications. In Table I, we use $A(i,:)$ to represent the dimension where elements are selected with a fixed index i. For example, the MTTKRP tensor operation in Table I is widely used in tensor factorization (e.g., recommended system); Stencil is an operation that updates the original matrix by accumulating neighboring elements; in element-wise matrix-matrix multiplication (EWMM), elements from two equal-size matrices in the same position is multiplied without accumulation; Khatri-Rao product (KRP) is the operation without accumulation where each element in one matrix is multiplied with all elements in another matrix.

The real-world tensors can be involved with different mixes of dense and sparse representation. For example, the tensor in MTTKRP is naturally with high sparsity. The sparsity in deep learning algorithms is caused by the nonlinear operator rectified linear unit (ReLU) function and model pruning. Table I shows the computation of sparse matrix-matrix multiplication (SpMM) in compressed sparse row (CSR) format [4]. CSR is widely used to store sparse matrices where a matrix is represented by three 1-D arrays: 1) row pointers to record the number of nonzeros from the first row to the ith row (A_{ptr}); 2) column indices to record the column index of each nonzero (A_{idx}); and 3) values to record the value of each nonzero (A_{val}). There are other formats like compressed sparse column (CSC) and coordinate list (COO) to store sparse matrices [4].

The variation in tensor operation renders hardware acceleration difficult. Although most tensor computation is composed of a series of MACs or only multiplications, the parallel pattern and accumulation pattern of partial sums vary in dimensions and size. Besides, the tensor can be sparse in real-world applications where the sparsity may differ in orders of magnitude. Finally, the difference in size and computation

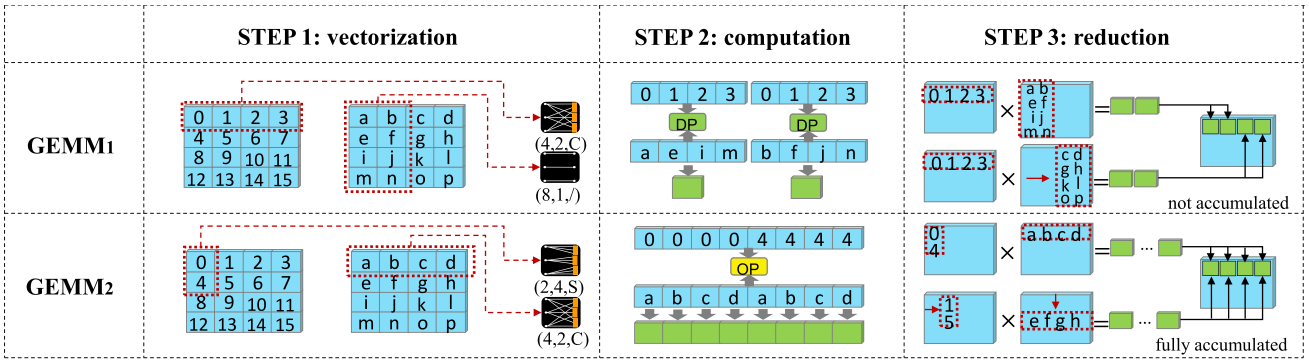


Fig. 2. Examples of execution model in three steps. In this figure, we only depict (size, replica, vec_type). C: circular manner. S: sequential manner.

leads to different data reuse opportunities and preference of parallelization, making it hard to accelerate tensor operations with a fixed execution model.

III. Flexible Execution Model

A. Execution Model Design

The fundamental component of Morphling is a flexible execution model that can support various tensor operations. Depending on how the data are vectorized and reduced to the final output, we generalize the tensor execution model in three steps.

Step 1 (Vectorization): In this step, the input tensor is transformed into a vector. During transformation, the input tensor can be duplicated either in a circular or sequential manner

$$\text{in\_vec} = \text{vectorize}(\text{src}, \text{size}, \text{replica}, \text{vec\_type}) \quad (1)$$

where src is the input tensor, size is the size of input tensor, replica represents how many times these elements are duplicated, vec_type can be either circular or sequential. For example, vectorizing (x_1, x_2, x_3) to $(x_1, x_1, x_1, x_2, x_2, x_2, x_3, x_3, x_3)$ is in sequential manner with replica = 3.

Step 2 (Computation): In this step, the transformed vectors are multiplied together following different parallel patterns. The computation step can be formulated as follows:

$$\text{out\_vec} = \text{compute}(\text{in\_vec}_1, \text{in\_vec}_2, \text{length}, \text{para\_type}) \quad (2)$$

where in_vec$_1$ and in_vec$_2$ are the vectorized tensors with the same length, para_type is the type of parallel pattern.

The computation step has the flexibility to parallelize the multiplications in different ways. Depending on the number of inputs and outputs that are linked together in the computation step, we define three parallel patterns, including many-to-one, one-to-many, and one-to-one. The many-to-one pattern refers to the operator where a single output depends on multiple inputs. The one-to-many pattern refers to the operator where a single input element is used for multiple output elements. The one-to-one pattern refers to the operator where a single output depends solely on a single input element.

For dense tensor computation, we implement these three parallel patterns using DP, OP, and EWVM operators as shown in Table II. DP operator corresponds to many-to-one pattern where the partial sums are accumulated together to yield a single output element. For example, in GEMM, the intermediate results of multiple multiplications are accumulated together in the dimension "K" using DP. OP corresponds to one-to-many pattern where output elements are generated by multiplying the element in one vector with all other elements in another vector. For example, in the KRP operation, each element in one matrix is multiplied with all elements in the other matrix. EWVM corresponds to one-to-one pattern, e.g., EWMM operation. The operators for sparse tensor operations will be introduced in Section III-C.

TABLE II
Dense and Sparse Operators for Parallel Patterns

Parallel Pattern	Many-to-one	One-to-many	One-to-one
Dense model	Dot product	Outer product	EWVM
Sparse model	Row-wise product dense×sparse	Kronecker product dense⊗dense	Block-wise sparse⊙sparse

Step 3 (Reduction): The output vector from the computation step could be the partial sums of the final result. Therefore, this step accumulates the output vectors from multiple computation steps to generate the final output

$$\text{rst} = \text{reduce}(\text{out\_vec}_1, \text{out\_vec}_2, \text{length}, \text{type}, \text{start}, \text{end}) \quad (3)$$

where out_vec$_1$ and out_vec$_2$ are output vectors from the computation step, start and end are used to specify the range if partial accumulation (PA) is needed, type is one of three accumulation types, including full accumulation (FA), PA, or no accumulation manner (NA). GEMM$_2$ is an example of FA, and 2D-CONV$_2$ is an example of PA.

These three steps are tightly correlated. The computation step determines how the input tensors are transformed in the vectorization step and how the output tensors are accumulated in the reduction step. Table I lists the three steps for various tensor operations. Fig. 2 presents two examples of using this execution model for GEMM, where we show how to use DP (GEMM$_1$) and OP(GEMM$_2$) for GEMM. For example, in GEMM$_2$, the elements from tensor A are duplicated in a sequential manner, and the elements from tensor B are duplicated in a circular manner to form an OP in the computation step. Different output vectors are accumulated using the FA pattern.

B. Format for Sparsity

For sparse tensor operation, the computation is tightly coupled with the compression format. Block compress sparse row

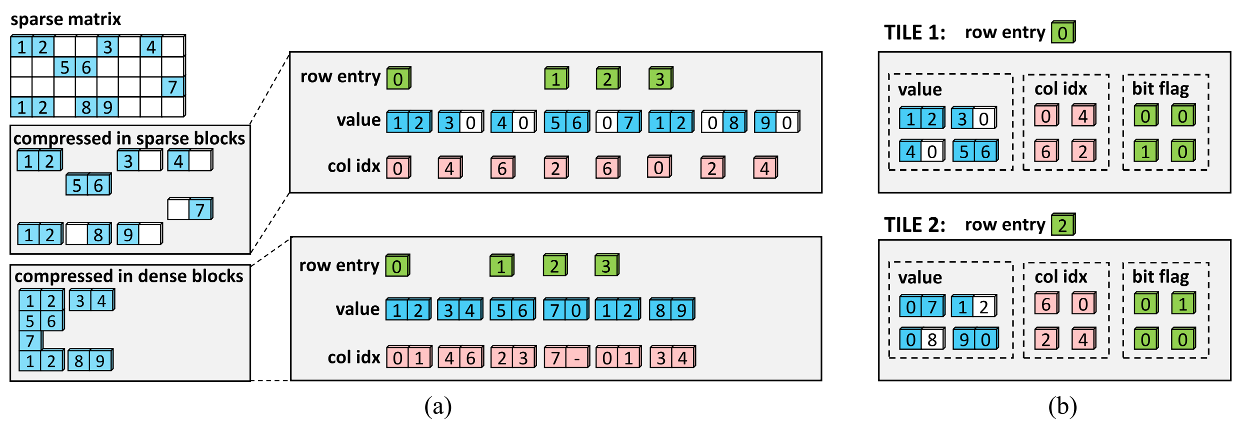


Fig. 3. (a) Tiled-BCSR format with nonzeros stored in dense blocks and sparse blocks. (b) Example tiled-BCSR using sparse block.

TABLE III
SPARSE TENSOR REPRESENTED IN TILED-BCSR FORMAT. WE USE (:,:) TO DENOTE THE DIMENSIONS THAT REPRESENTED IN TILED-BCSR FORMAT. "×": ROW-WISE PRODUCT; "⊗": KRONECKER PRODUCT; AND "⊙": BLOCK-WISE MULTIPLICATION

Tensor apps	**Dimensions in tiled-BCSR**	**Unified computation**	**Block dependency**
SpMM	A(:,:), B(:,:)	$Y(:,:) = A(:,:) \times B(:,:)$	multiple block-Bs $\rightarrow$ one output row
Sp2D-CONV	A(:,:,p,q), B(:,:,j+q)	$Y(:,j,:) = \sum_p \sum_q A(:,:,p,q) \times B(:,:,j+q)$	multiple block-As $\rightarrow$ one output row
SpKRP	A(:,:), B(:,:)	$Y(:,:) = A(:,:) \otimes B(:,:)$	one block-A $\rightarrow$ multiple outputs
Sp3D-CONV	A(:,:,r), B(:,:,k+r)	$Y(:,:,k) = \sum_r A(:,:,r) \times B(:,:,k+r)$	multiple block-As $\rightarrow$ one output row
SpEWMM	A(:,:), B(:,:)	$Y(:,:) = A(:,:) \odot B(:,:)$	1 block-A $\rightarrow$ 1 block-B
SpMTTKRP-S1	A(i,:,:), B(:,:)	$Y_1(i,:,:) = A(i,:,:) \times B(:,:)$	multiple block-Bs $\rightarrow$ one output row
SpMTTKRP-S2	Y_1(:,:), C(:,:)	$Y(i,:) = \sum_j Y_1(i,:,j) \odot C(:,j)$	1 block-Y_1 $\rightarrow$ 1 block-C

(BCSR) format is a structural representation and allows continuous data access within a block. Previously, BCSR format has been used in software library in HPC domain for specific kernel such as SpMV [7], [8]. However, the real-world tensor often shows the irregular distribution of nonzeros, which makes the hardware suffer workload imbalancing problem. To address this, we propose a tiled-BCSR format where nonzeros are first packed into blocks and then organized into tiles. Compared with traditional formats, such as CSR, CSC, tiled-BCSR format shows regular accumulation, and data access patterns. Besides, sparse tensor operations can be parallelized in multiple tiles, which helps to balance the workload and provide high parallelism.

We first design two types of blocks that are: 1) dense block and 2) sparse block as shown in Fig. 3(a). As the computation step only involves a vector operator, we restrict the block shape as a vector. In a dense block, the entire row is first compressed into a dense vector by eliminating the zero elements and then divided into blocks. In a sparse block, the entire row is first divided into blocks and then compressed. The dense block is designed to increase the utilization of multipliers. The sparse block has a low overhead of locating the output address since the column indices are continuous within a block.

To achieve a balanced workload for each PE, the blocks are batched into multiple tiles with each tile contains multiple continuous rows, as shown in Fig. 3(b). Each tile has a row entry information which indicates the starting row index of the tile. For each tile, there is a bit-flag matrix to record the row index change where "1" means this block is in the next row. To calculate the row index, we only need to insert a bit counter logic in each PE.

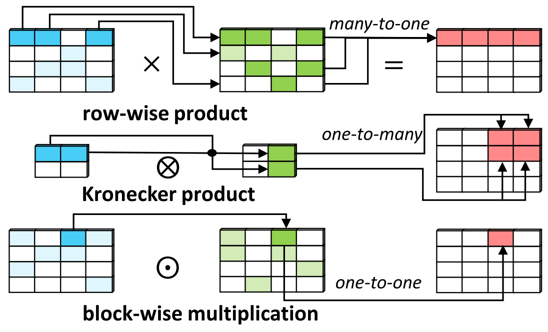


Fig. 4. Operators for the sparse computation step.

C. Enabling Sparse Computation

In our execution model, the parallel patterns are unified, while the meta-operator is different, as shown in Table II. Both sparse and dense operation can be executed on our unified architecture design. To establish a general sparse tensor execution model, we begin with a tensor operation whose input tensors are represented as a set of matrices stored in tiled-BCSR format. For example, in Table III, the 4-D tensor A in 2D-CONV is compressed in the last two dimensions with the first two dimensions stored as a pointer.

To support the parallel patterns in the computation step in Section III-A, we define three operators for sparse tensor computation as shown in Fig. 4. In the row-wise product operator $\times$, each block in matrix A (denoted as block-A) is required to be multiplied with several blocks in different rows of matrix B (denoted as block-B), where the row entry of block-B is determined by the column index of the nonzero value from block-A [24]. This corresponds to the many-to-one pattern, as multiple rows of matrix B are calculated for one row in the result matrix. The Kronecker product operator ($\otimes$) is a generalization of the OP from vectors to matrices, which corresponds to the one-to-many pattern. Block-wise multiplication operator $\odot$ is similar to EWVM pattern [75].

Using these operators, the sparse tensor computation can be represented as follows:

$$Y(:,:) = \mathbf{M(n\text{-}1)} \ \triangle \ \mathbf{M(n\text{-}2)} \ \triangle \cdots \mathbf{M(2)} \ \triangle \mathbf{M(0)}. \quad (4)$$

$M(i)$ is defined as a tiled-BCSR matrix selected from the ith input tensor. The output $Y(:,:)$ can be a vector or a matrix of the final results. $\triangle$ is the operator of two tiled-BCSR matrices. Table III shows how the sparse tensor operations are represented using (4). For example, MTTKRP can be represented as $Y(i,:) = \sum_j A(i,:,j) \times B(:,:) \odot C(:,j)$.

Depending on the parallel pattern shown in Table II, the matrices can be either encoded using dense block or sparse block. In a row-wise product, multiple blocks of matrix B may link to the same output element if they share the same column index. Therefore, we store matrix B in sparse block to keep the column index continuous and matrix A in dense block to maintain enough workload for multipliers, which achieves a balanced tradeoff between hardware efficiency and indexing overhead. As the Kronecker product does not require index matching, both two matrices are stored in dense blocks to maximize the utilization of multipliers. In block-wise multiplication, the indices of blocks from two matrices need to be strictly the same. The two matrices are stored in sparse blocks to reduce index comparison.

Table III also gives the dependency between the blocks of two tiled-BCSR matrices for different tensor computation. For row-wise product, as one block in matrix A is linked to multiple blocks in matrix B, therefore, elements in block-A are replicated multiple times in the vectorization step. And, the computation step needs to accumulate the results that share the same column index in the block-B. Similar to the outer product, the Kronecker product operator replicates values in both matrices in the vectorization step. And, it is a one-to-many operator as one input block is linked with multiple elements in the output. Block-wise multiplication does not require the vectorization step, and the blocks with the same row entry and column index will be multiplied in the computation step.

IV. Architecture Design

Morphling is a tiled architecture consisting of a reconfigurable PE array and a reconfigurable communication network. Fig. 5 presents the overview of Morphling architecture where each PE contains a local memory for data storage and a router to transfer data. Each switch contains buffers to exploit data reuse and ALs to support accumulation in the reduction step. The on-chip scratchpad interfaces with the DRAM through multiple channels.

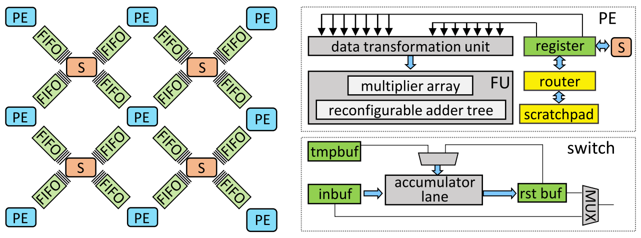


Fig. 5. Architecture overview. PE: processing element. S: switch. FU: function unit. The FIFO is used to transfer data between PEs and switches.

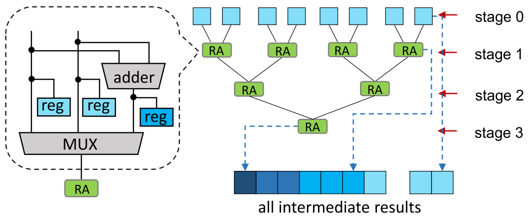


Fig. 6. Architecture details of reconfigurable adder tree.

A. PE Design

The vectorization and computation steps in the tensor execution model are implemented within a PE. In other words, each PE vectorizes the input tensor and performs one of three basic vector operations in the computation step. As shown in Fig. 5, the data vectorization unit (DVU) is used to vectorize the selected elements into two vectors with the same length. The function unit (FU) will first check whether multiplication operations are needed (if it is DP or OP) for the tensor operation. Then, it sends the results to an adder tree to generate the output vector.

Each PE features three kinds of reconfigurability. First, the input tensor can be loaded either from the global scratchpad via the router, or from the switch buffer via the local FIFOs. Each PE has two input tensor buffers, and each buffer interfaces with a control unit that can be dynamically configured to switch the source of the input tensor. Second, the DVU has the flexibility to support different vectorization. The vectorization step can be different in terms of tensor size and duplication type, which is configured by the instructions. Third, the FU has the flexibility to support different vector computation. The FU includes a multiplier array and a reconfigurable adder tree. The adder tree can support DP operations with different lengths. Besides, if the tensor is sparse and stored in a compressed format, then addition operations will be performed according to the compressed index. Special instructions are needed to configure the PE for different vector computation and the configuration information is stored in registers.

B. Reconfigurable Adder Tree

Fig. 6 shows the micro-architecture of the reconfigurable adder tree with eight inputs. The reconfigurable adder tree receives multiplication results from the multiplier array. The tree is divided into multiple stages where the intermediate result from each stage is stored separately in registers. Each reconfigurable adder receives two inputs and sends one output

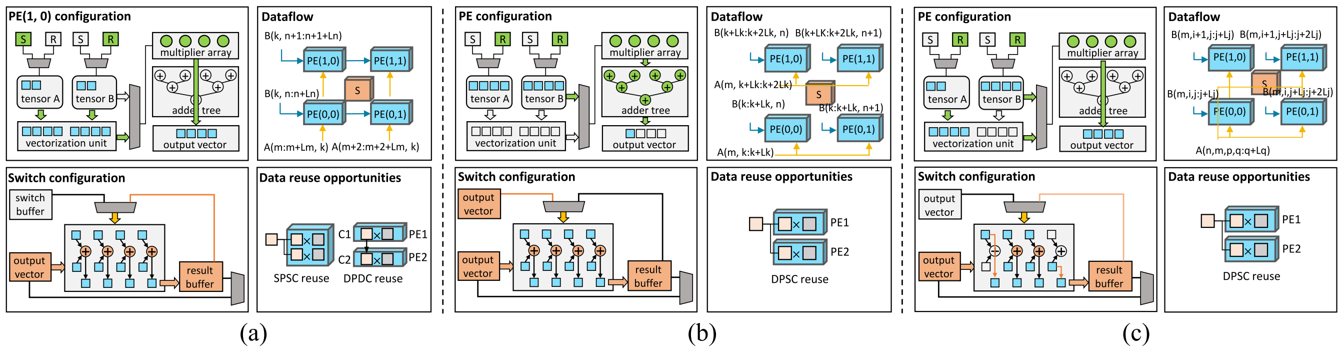


Fig. 7. Architecture configurations for GEMM and 2D-Conv. We use **Lx** to represent the parallelization degree in the dimension **x**. White color means not activated. S and R mean switch and router. (a) GEMM. PE : OP op. Switch: iterative accumulation. (b) GEMM. PE : DP op. Switch: forward accumulation. (c) 2D-CONV. PE : EW op. Switch: forward accumulation with shift.

forward to the next adder. The forward element is selected among the two inputs and their addition result via a multiplexer (MUX). When accelerating dense tensor operations, the adder tree is symmetrically configured for DP parallelization. In the example of GEMM$_1$, the adders in stages 1 and 2 are activated to form two DP operations. Similarly, in 2D-CONV$_1$ example, only the adders in stage 1 are activated. When handling sparse tensor operation, each adder is dynamically reconfigured according to indices. Details of sparse optimization is discussed in Section IV-F.

C. Switch Design

In the computation step, each PE generates one output vector. The reduction step is an inter-PE operation implemented using switch. The switch has the flexibility to support either iterative accumulation or forward accumulation pattern, as shown in Fig. 7. Iterative accumulation pattern, as shown in Fig. 7(a) gathers the output vectors from the same PE, which only has a single input source and iteratively accumulates the output vectors at different cycles. Forward accumulation, as shown in Fig. 7(b) is an accumulate-and-forward chain to accumulate output vectors from different PEs. To ensure the accumulation pattern can be alternated, both data from adjacent PEs and the result in the last cycle are buffered in the local memory. There is a selector to distinguish whether it is the output vector from the PE or the previous result in the local buffer. Besides, the AL can shift the output vector with a given length to enable PA in the reduction step.

On the other hand, the switch can be used as a bridge for nPE data communication by gating the AL. As shown in Fig. 5, the switch is employed to enable data sharing among PEs. Each switch is connected with four adjacent PEs using simple FIFO logic. The FIFO can either send the input tensor or the output vector from PEs to the switch. Similarly, a configurable multiplexer, connected with input buffer and result buffer of the switch, controls which data is required to send back to the PE array.

D. Reuse Analysis

Morphiling architecture exploits three types of data reuses. First, there exists data reuse in a single PE, named same PE same cycle reuse (SPSC). The SPSC reuse is from the vectorization step in the execution model. It reuses the input tensor at the register level, where the PE duplicates the tensor and stores it in local registers. Moreover, the data reuse may arise from different PEs, which can be further categorized into different PE same cycle reuse (DPSC) and different PE different cycle reuse (DPDC). The DPSC reuse means that the computation steps in different PEs share the same input tensor at the same cycle. The DPSC reuse is exploited by reading the tensor from the buffer only once and broadcasting the data to multiple PEs via routers. Our switch design supports the DPDC reuse among PEs. Clearly, the switch enables both multicycle input reuse or output reuse by buffering the intermediate data within two cycles. Multicycle output data reuse is achieved by making output results stationary in the switch and iteratively updating the input buffer in different cycles. The input data reuse is enabled by configuring the multiplexer to directly transfer the input data to another PE, as shown in Fig. 5. The multicycle input data reuse is also supported between distant PEs by connecting multiple switches and configuring the datapath. Both SPSC and DPSC are spatial data reuse, while the DPDC reuse is temporal.

E. Architectural Examples

We use three examples to illustrate our reconfigurable architecture. Fig. 7(a) applies OP to GEMM using output-stationary dataflow [13], [40], [69]. We depict the detailed configuration of PE$_{(1,0)}$ and switch. This PE is configured to access one input tensor from PE$_{(0,0)}$ via switch and the other input tensor from scratchpad via router. For this dataflow, output vectors from the same PE need to be accumulated. Therefore, the switch is configured as iterative accumulation to generate the final result. Meanwhile, SPSC and DPDC reuses are exploited. Fig. 7(b) applies DP to GEMM using input-stationary dataflow [13], [40], [69]. The input tensor is directly sent to the multiplier array with the vectorization unit skipped. Besides, the output vectors from the PE$_{(0,0)}$ and PE$_{(1,0)}$ are accumulated using forward accumulation configuration of the switch. Fig. 7(c) uses row-stationary dataflow [12] for 2D-CONV. The convolution filter slides with overlap and thus the output vectors from PEs are partially accumulated in the AL. In Fig. 7(b) and (c), DPSC reuse is exploited.

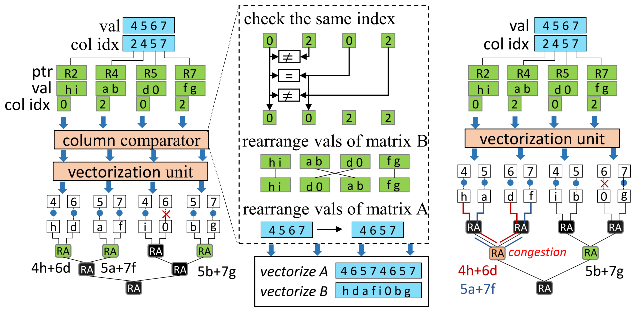


Fig. 8. Working flow example of row-wise product using tiled-BCSR format and the congestion example. The blue block is from matrix A and green blocks are from matrix B. (a) Working flow for sparse tensor operation. (b) Example of congestion.

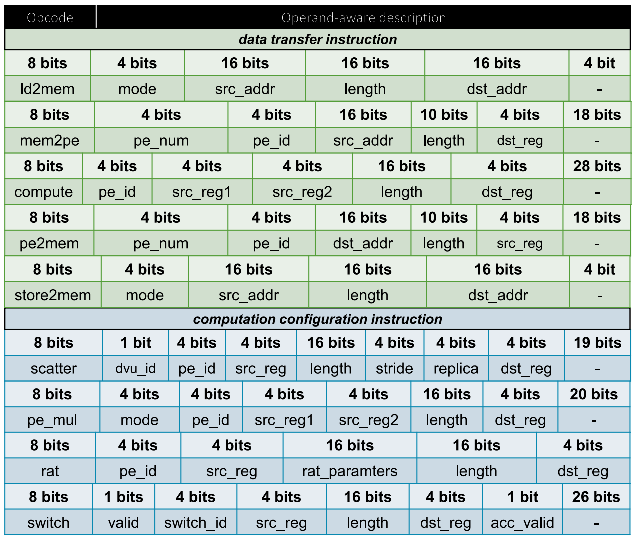
F. Sparse Computation Optimization

Though a unified computation pattern for sparse tensor operations is proposed, an inefficient hardware implementation may cause an accumulation congestion problem. Fig. 8(b) shows the datapath of row-wise DP that exists accumulation congestion. The multiplication results $4h$ and $6d$ from different block-B is required to be added, however, distributed in the different adders at the first stage. This phenomenon also happens to the results $5a$ and $7f$. Therefore, the adder in the next stage has to finish two addition operations which can decrease the hardware efficiency. To handle the congestion problem, we design a column index comparator that rearranges the data in a more hardware-efficient manner as shown in Fig. 8(a). The column index comparator checks the column index of block-B and sets the values continuously whose column indices are the same in block-B. By doing this, the accumulation congestion is reduced. Note that the values in block-A and block-B also need to be rearranged according to the column index comparator, as shown in Fig. 8.

G. Application Mapping

There exist different parallelization strategies for the same tensor application. Different dataflow results in different configurations of the Morphling architecture which affects data reuse, execution latency, and energy cost. We begin with a tensor application represented as a loop-based iteration domain where each node is one loop instance in the original code.

We use the polyhedral model to select different schedulings to map tensor applications on Morphling. The polyhedral model provides powerful abstractions to optimize loop nests with regular accesses and captures a complex sequence of loop transformations [5]. The objective function aims at minimizing latency and energy cost. The constraints consist of resource constraints, bandwidth constraints, and energy budget. The resource constraints include the number of multipliers that determines the node number that can be executed in parallel and the length of the AL which determines how many partial sums can be accumulated together. The bandwidth constraints limited the number of nodes accessed in parallel which further affects the time scheduling function. In summary, the optimization problem turns to be an integer linear programming (ILP) problem. Then we enumerate the solutions whose resource or bandwidth utilization is lower than 30%, and choose the solution with minimum latency.



Opcode	Operand-aware description							
data transfer instruction								
8 bits	4 bits	16 bits	16 bits	16 bits	4 bit			
ld2mem	mode	src_addr	length	dst_addr	-			
8 bits	4 bits	4 bits	16 bits	10 bits	4 bits	18 bits		
mem2pe	pe_num	pe_id	src_addr	length	dst_reg	-		
8 bits	4 bits	4 bits	4 bits	16 bits	4 bits	28 bits		
compute	pe_id	src_reg1	src_reg2	length	dst_reg	-		
8 bits	4 bits	4 bits	16 bits	10 bits	4 bits	18 bits		
pe2mem	pe_num	pe_id	dst_addr	length	src_reg	-		
8 bits	4 bits	16 bits	16 bits	16 bits	4 bit			
store2mem	mode	src_addr	length	dst_addr	-			
computation configuration instruction								
8 bits	1 bit	4 bits	4 bits	16 bits	4 bits	4 bits	4 bits	19 bits
scatter	dvu_id	pe_id	src_reg	length	stride	replica	dst_reg	-
8 bits	4 bits	4 bits	4 bits	4 bits	16 bits	4 bits	20 bits	
pe_mul	mode	pe_id	src_reg1	src_reg2	length	dst_reg	-	
8 bits	4 bits	4 bits	16 bits	16 bits	4 bits			
rat	pe_id	src_reg	rat_paramters	length	dst_reg			
8 bits	1 bits	4 bits	4 bits	16 bits	4 bits	1 bit	26 bits	
switch	valid	switch_id	src_reg	length	dst_reg	acc_valid	-	

Fig. 9. Architecture instructions.

H. Architecture Instructions

We design a set of instructions which are used to configure the Morphling architecture. Fig. 9 summarizes instructions for data transfer and computation configuration. Each instruction is aligned to 64 bits which are sufficient to support reconfigurable features and data address. Data transfer instruction supports variable data size across different dimensions. The off-chip data transfer instructions have three modes. Vectors and matrices are two commonly used tensor types which are specified as two data transfer modes. For the tensor that has higher dimensions, the instruction can load length elements from a given dimension and offset. *mem2pe* and *pe2mem* are used to transfer data between on-chip scratchpad and registers of PEs. Computation configuration instructions help to configure the unit in each PE so that the PE can perform specified tensor execution dataflow.

V. EXPERIMENTS

A. Methodology

Morphling Configuration: The Morphling architecture is organized as an 8×8 PE array and an 8×8 switch array. The target data type is 16-bit fixed point.Each PE has 16 multipliers with a 16-input reconfigurable adder tree. The scratchpad for input and output tensor is a 512×16 bit SRAM in the PE, and the scratchpad in the switch is a 256×16 bit SRAM. The design is written in the Chisel hardware description language [3]. We use Chisel to generate Verilog RTL. Then, we use Synopsys Design Compiler to estimate the chip area and total power under the TSMC 28 nm technology. The synthesized frequency is 600 MHz. Theoretically, the peak performance of Morphling is $64 \times 16 \times 0.6 = 0.61$ TOP/s. Table IV provides the detailed area and power breakdown for Morphling at a total area of 8.62 mm^2 and total power

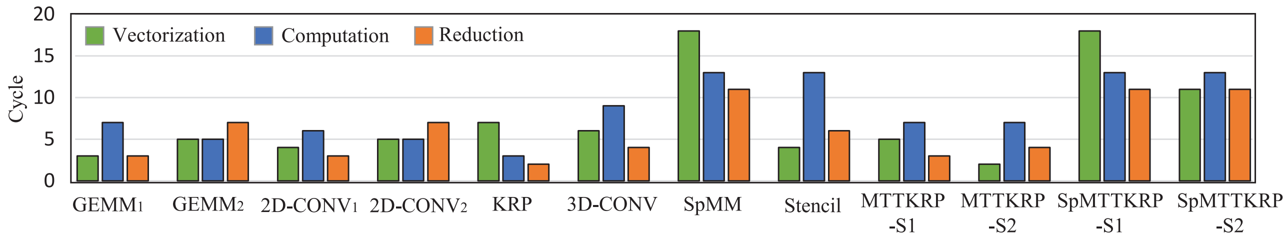


Fig. 10. Latency result of each step. SpMTTKRP is divided into two stages denoted as S1 and S2.

TABLE IV
MORPHLING AREA AND POWER BREAKDOWN

	Component	Area (mm^2)	Area (%)	Power (mW)	Power (%)
PE	Router	0.005	4.5	0.72	7.0
	DVU	0.036	34.0	0.22	2.1
	FU	0.031	29.1	3.68	35.8
	Memory	0.034	32.3	5.66	55.1
	Total	0.105	100	10.3	100
Switch	AL	0.013	42.8	3.91	45.4
	Memory	0.017	57.8	4.70	54.6
	Total	0.030	100	8.61	100
Morphling	64×PEs 64×Switches	8.62 mm^2		1.21 W	

TABLE V
BENCKMARKS OF DIFFERENT TENSOR OPERATIONS

Application	Domain	Tensor operation	Data size
Resnet[31]	DNN	2D-CONV GEMM	25M params 3.8G ops
ALS[6]	Matrix fabrication	MTTKRP	480K×18K×2K (Netflix)
SVM[21]	Classifier	GEMV	26K samps, 512 dim 128 categories
SpResnet[31]	DNN	SpMM	91.3% sparsity

of 1.21 W. To evaluate the performance of Morphling, we developed a cycle-accurate model based on Chisel. All the required data are initially stored in DRAM where the sparse matrices are stored in tiled-BCSR format. Morphling interfaces with DRAM through multiple DDR channels. The DRAM bandwidth is assumed to be 40 GB/s which provides ample bandwidth for most tensor applications. For the input data size that is larger than the size of the on-chip scratchpad, we divided the data into multiple tiles. For DRAM simulation, we measure the data size and the number of DRAM access from the Chisel tester. We build our polyhedral model based on the integer set library (ISL) [45].

Other Platforms: On GPUs, we run tensor applications on Titan X using CuBLAS 10.0 [58] for dense tensor operations, CuSPARSE 10.0 [60] for sparse tensor operations and CuDNN 6 [59] for deep learning applications. On CPUs, we evaluate tensor operations on Intel processor i7-9700k using Pytorch [67]. On FPGAs, we run experiments on Xilinx ZC706 FPGA. Our FPGA implementation is operated at 166-MHz frequency. The benchmark is implemented in OpenCL code and synthesized using Xilinx SDx 2018.2 [86]. On tensor processing units (TPUs), we build a TPU-like platform for comparison. We model a 32 × 32 systolic engine which has the same computational ability as Morphling. The estimated power is 0.78 W using Synopsys Design Compiler under the same technology and synthesis frequency as Morphling.

Benchmark: We first use the tensor algebra in Table I. Then, we evaluate four real-world tensor applications as shown in Table V. Resnet is a deep neural network (NN) for image recognition which consists of many residual blocks [31]. We then prune Resnet using the technique in [27] and [28]. As a result, we achieve 91.3% weight sparsity, and the average sparsity of input images 43%. We transform the sparse Resnet into a series of SpMM and use it to evaluate the performance of Morphling. Alternating least squares (ALSs) are most widely used method for canonical polyadic decomposition (CPD) where MTTKRP is a core operation. We use Netflix as the data set which is taken from Netflix Prize competition [6].

B. PE Latency Profiling

Fig. 10 presents the cycle latency of three steps in Morphling architecture. The latency of each step varies for different tensor algebra due to different hardware configurations. For dense tensor operations, the latency of the computation step mainly depends on the activated stage in the adder tree. The latency of the reduction step is affected by the data transfer between PEs and the configuration of ALs in the switch. GEMM, 2D-CONV, and 3D-CONV are computation-intensive operations and there exists data reuse opportunities. GEMM_1, 2D-CONV_1, and 3D-CONV apply DP parallelization with the input tensor broadcast to multiple PEs. Therefore, they need extra cycles in the computation step as the adder tree is activated. On the other hand, GEMM_2 and 2D-CONV_2 are parallelized using OP, which exploits SPSC reuse as the input tensor is duplicated in the vectorization step. They also feature DPDC reuse as they are implemented in a systolic array by configuring the switch. They show lower latency in the computation step but higher latency in the reduction, as the adder tree is inactivated while the ALs are configured to gather the output. KRP has no addition operation thus shows the lowest latency of the computation step and reduction step.

The latency of three steps for sparse tensor operations is higher due to format decoding and data rearrangement. As shown in Fig. 8(a), the vectorization involves column index comparison to avoid congestion, which costs extra cycles. Besides, the reconfigurable adder tree is dynamically configured to determine the forward data, which leads to higher

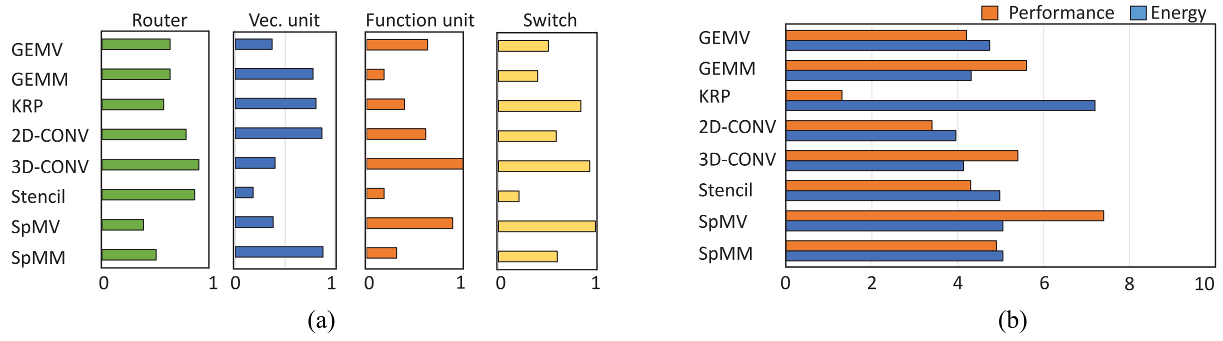


Fig. 11. Energy breakdown and comparison with FPGA. (a) Normalized energy consumption. (b) Speedup and energy efficiency over FPGA.

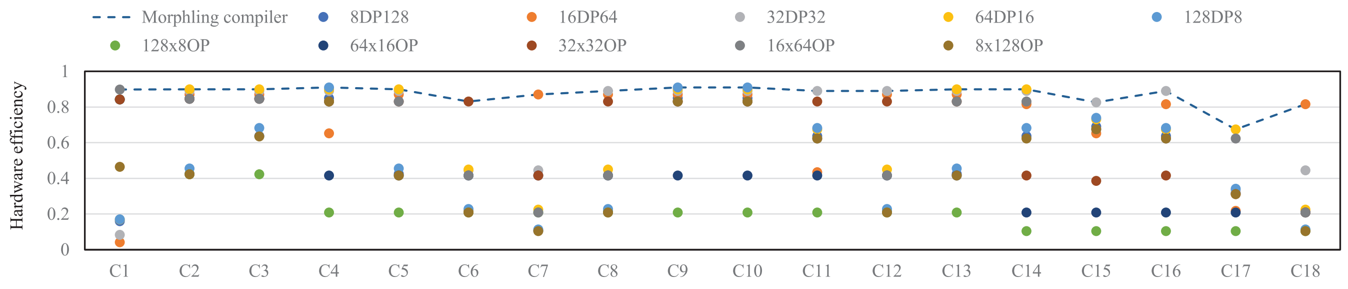


Fig. 12. Normalized hardware efficiency of different parallel pattern on the first 18 layers of GoogLeNet. *a*DP*b* means *a* DP operations with the vector size *b*. *a*x*b*OP means OP operation with one vector size is *a* the other is *b*.

latency in the computation step. For the reduction step, the gap between dense and sparse operation is small as the indices within a sparse block are continuous. SpMTTKRP-S2 applies a block-wise multiplication parallel pattern, which has low complexity. Therefore, it shows the lowest latency in the vectorization step compared with other sparse operations.

C. PE Energy Breakdown

Fig. 11(a) reports the energy breakdown for the modules in the PE, including routers, vectorization unit, FU, and switch. The energy cost of the vectorization unit and FU mainly depends on the computation complexity. The data transfer between the PE array and on-chip scratchpad affects the energy cost of routers. The energy cost of switch results from two aspects: 1) buffer controller to enable data transfer between PEs and 2) ALs to support different accumulation patterns.

GEMV, KRP, and Stencil are communication-intensive operators. These operations show low utilization of computing resources, such as FU and the switch. Stencil requires more switch resources because the switch array is configured as a systolic array. Fig. 11(b) shows the performance and energy comparison results over the FPGA platform. We observe that Morphling can achieve higher speedup and energy efficiency for those computation-intensive operations, such as GEMM, 2D-CONV, and 3D-CONV. These operations show a high resource utilization. Overall, we achieve 1.1X–7.4X performance speedup and 1.3X–37X energy efficiency over FPGA. This benefit comes from the architectural advantage of our reconfigurable design.

D. Dataflow Optimization

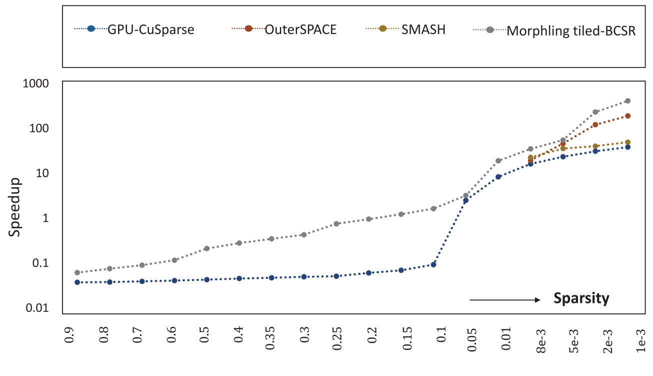
There exists a design space composed of different parallel pattern choices for tensor applications. For the same tensor operation, the optimal dataflow varies as the input tensor shape and size change. The flexible execution model of Morphling opens up the opportunities for design space exploration of different parallel patterns.

We use the first 18 layers of GoogLeNet with varied tensor sizes to evaluate the efficiency of different implementation choices. These layers are composed of 2D-CONV, GEMM (transformed from the convolution with 1×1 filter) and Pooling (down-sampling operation without multiplication), which have different behaviors. Fig. 12 shows the results. The optimal configuration varies across different layers resulting from the diverse dimension size of the involved tensor. For example, layer "C2, C3, C5, and C8" show higher hardware efficiency by using DP parallelization because the size of the parallelized dimension is large. Thanks to the polyhedral mapping, Morphling always chooses the best parallel pattern.

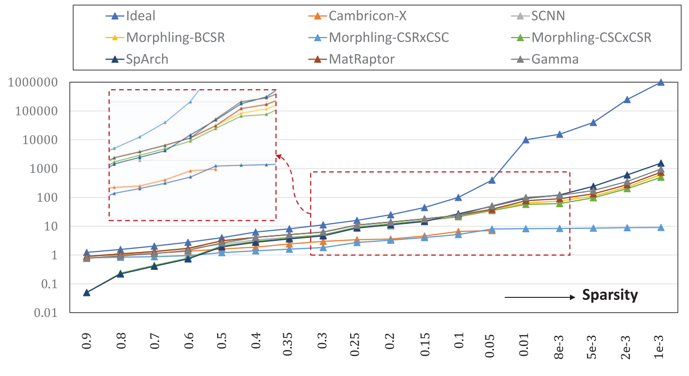
E. Efficiency for Sparsity

Fig. 13 shows the comparison with GPU, Cambricon-X [89], OuterSPACE [61], SCNN [62], and SMASH [35]. The baseline of GPU-CuSPARSE, OuterSPACE, and SMASH is set as dense GEMM in GPU-CuBLAS. The baseline of Cambricon-X and SCNN is set as the dense version of their architecture since these accelerators support both dense and sparse operations. We also draw the line of theoretical speed up calculated by (1/sparsity).

Comparison With GPU: Compared with GPU-CuBLAS, when the sparsity is higher than 0.05, GPU-CuSPARSE shows lower performance due to the memory uncoalesing problem and workload imbalance problem. Morphling inherently supports sparse computation by extending the parallel pattern to tiled-BCSR format under a unified execution model. More importantly, tiled-BCSR format helps to balance the workload.



(a)



(b)

Fig. 13. Speedup of Morphling over GPU and state-of-the-art accelerators using synthetic matrices with sparsity (the sparsity is defined as the proportion of nonzeros) varying from 0.9 to 0.001. The data of SCNN [62], Cambircon-X [89], SMASH [35], and OuterSPACE [61] are from the original paper. The data of SpArch [90], MatRaptor [72], and Gamma [88] are obtained by simulator. The data of GPU-CuBLAS and GPU-CuSPARSE are measured on TitanX and obtained using NV profiling. (a) Baseline with baseline with GPU-CuBLAS. (b) Baseline with the dense version of the architecture.

Therefore, Morphling can achieve nearly ideal performance when the sparsity is high, and achieves around 10.8X speedup compared with GPU-CuSPARSE when the sparsity is lower than 0.01.

Comparison With ASIC Accelerators: Morphling shows 1.6X–8.4X speedup and 1.2X–2.2X compared with SMASH [35] and OuterSPACE [61] when the sparsity is lower than 0.01. SMASH uses a software encoding scheme based on a hierarchy of bitmaps. This format shows highly efficient indexing for output address, however, requires a decoding module. OuterSPACE applies OP dataflow and has a long linked list of partial sums that requires index sorting, while the column comparator of Morphling has fewer inputs and only checks equality. When the sparsity is high, Morphling shows similar speedup compared with SCNN [62] and Cambricon-X [89]. When the sparsity reduces to 0.2, Morphling outperforms SCNN and Cambricon-X with 1.4X and 3.9X speedup. SCNN uses the format that stores the number of nonzero values followed by the number of zeros before each value. This format incurs a high overhead in the computation of the output address. And, Cambricon-X uses DP dataflow which requires index comparison leading to low multiplier utilization. We also build an analytical simulator to estimate the speedup of three state-of-the-art sparse accelerators, SpArch [90], MatRaptor [72], and Gamma [88]. SpArch [90] applied an OP-based approach with a Huffman tree scheduler to merge the results. When the sparsity is more than 0.1, SpArch shows less speedup compared to the traditional OP method (CSC × CSR). This is because the encoding overhead of the input matrix and merge overhead of the output matrix are large. Both MatRaptor [72] and Gamma [88] employed row-wise product (In Gamma, it is called Gustavson dataflow) that is similar to our approach. We observe that MatRaptor and Gamma exhibits a liitle higher speedup compared to Morphling due to architectural efficiency and format efficiency. Morphling is a much more general architecture that aims to handle various tensor algebra, and adopts a less complex format. The benefit of MatRaptor comes from its C^2SR format that improves the memory coalescing, and merge queues to gather results in parallel. On the other hand, Gamma proposed a new cache design that is specific to minimize memory traffic.

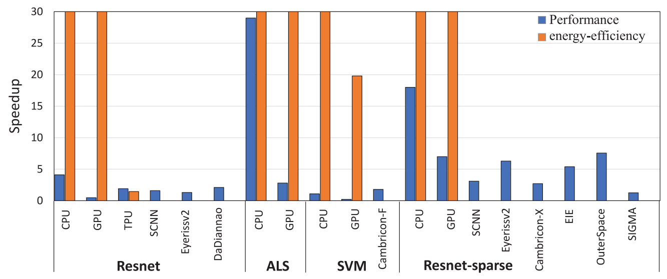


Fig. 14. Performance and energy-efficiency comparison using four tensor benchmarks.

Comparison With Traditional Formats: To demonstrate the efficiency of tiled-BCSR, we depict the efficiency of two cases using CSR and CSC format, where CSR(CSC) × CSC(CSR) means the matrices A and B are stored in CSR(CSC) and CSC(CSR) format. Tiled-BCSR format outperforms CSR × CSC approach with 1.1x–68.2x speedup. This is because CSR × CSC needs an index comparison operation which can lead to a low utilization of multipliers, while Morphling adopts row-wise product operation which does not require index comparison. On the other hand, the Tiled-BCSR format shows 1.2X–18.1X speedup compared with CSC × CSR approach. Though CSC × CSR approach shows no index comparison, it results in a large number of write operations with irregular output addresses. Another benefit of tiled-BCSR format is it has a balanced workload compared with CSR and CSC format.

F. Study of Real-World Tensor Applications

Here, we compare our design with state-of-the-art accelerators using four real-world tensor applications in Table V. Specifically, we use 1024-PE design in Eyeriss-v2 [13], 4 Cambricon-F cores in Cambricon-F [91], the original design in SCNN [62]. For other designs, we scale their design to 1024 multiplier and build a cycle-accurate model for evaluation. Fig. 14 shows the performance speedup and energy efficiency of Morphling over these accelerators. As shown in Fig. 14, Morphling can support a wide range of tensor applications. Compared with general processors, such as CPU and GPU, Morphling shows higher energy efficiency with comparable performance to GPU.

For Resnet, Morphling has attained 4.1X speedup and 677.7X energy-efficiency compared with CPU. The performance of Morphling is 0.47X of TitanX GPU due to the limited computing resource, but Morphling achieves 44.7X energy efficiency. Compared with TPU, SCNN-dense [62], Eyeriss-v2 [13], and DaDiannao [14], Morphling shows 1.9X, 1.6X, 1.3X, and 2.1X speedup, respectively. The resnet algorithm has different tensor operations and diverse dimension size. Morphling can flexibly configure the hardware for the tensor operation. Morphling shows 29X, 2.8X speedup and 4666.7X, 256.3X energy-efficiency compared with CPU and GPU, and Morphling achieves similar performance to Cambricon-F. For Resnet-sparse, we transform it to SpMM operation. Morphling achieves 18X and 7X speedup compared with CPU and GPU. We also compare to other sparse ASIC accelerators, including SCNN-sparse [62], Eyeriss-v2 [13], Cambricon-X [89], EIE [26], OuterSpace [61], and SIGMA [69]. The difference has been discussed in Section V-E. EIE [26] adopts CSC format where DP is performed in different PEs, leading to high PE communication overhead. Morphling shows similar performance compared to SIGMA, which also applies a reconfigurable adder tree to gather the results.

VI. Related Work

Coarse-Grained Reconfigurable Architectures (CGRAs): CGRAs has been developing rapidly since the 2000s and continue to attract increasing interest [15], [20], [36], [47], [55], [63]. Recently, the demand for massive parallel computation has grown continuously in the field of CGRAs [16], [66], [70], [77], [78], [87]. Plasticine [66] is a CGRA written in Chisel. At the architecture level, Plasticine is designed for general application, which is not tensor specific and lacks a flexible execution model to guide the hardware dataflow. Gorgon [78] and Capstan [70] are derivatives of the Plasticine, which are designed for enabling sparsity. Aurochs [77] is extended from Gorgon [78] where it introduces a threading model that extracts parallelism from irregular data structures. Thinker [87] can be reconfigured for Hybrid NNs that can process different layer types of NNs in parallel. Thinker only focuses on the CNN domain, while Morphling targets a wide range of tensor applications. Nowatzki *et al.* [57] also proposed an execution model using stream dataflow. However, this execution model is supported by designing peripheral control logic. In contrast, Morphling design a tensor-specific PE and ALs to support the execution model. Dadu *et al.* [16] proposed a reconfigurable systolic array that was composed of sparse processing units (SPUs). SPU is a general CGRA design which primarily focuses on stream-join control for general data dependency, while Morphling is a tensor-specific CGRA which targets applications with massive MACs.

Dense Tensor Accelerators: As many applications involve tensor computation, various tensor accelerators are designed for acceleration. However, these accelerators are usually focused on a single operation, e.g., convolution or matrix multiplication. TPU [34] is developed by Google for NNs processing in datacenters. TPU is a systolic data flow of the Matrix Multiply Unit. Cambricon [48] is a domain-specific instruction set architecture for convolution and GEMM. Diannao, Pudiannao, Shidiannao, and Dadiannao [11], [14], [18], [46] are a set of architectures extended from Cambricon as machine learning accelerators. DySER [23] is a tensor accelerator that features with functionality specialization and parallelism specialization. Similar to our reconfigurable tree, DySER contains a reduction tree. The difference is that our adder tree has additional logic that helps to handle irregular sparsity with high hardware efficiency. Convolution Engine [68] has the reconfigurability for different types of convolution. Convolution Engine also contains a flexible reduction tree to fuse multiple instructions, while the main goal of our reconfigurable adder is to provide support for sparsity.

Sparse Tensor Accelerators: References [13], [25], [26], [44], [50], [51], [89], [92] are sparse DNN accelerators. MAERI [41] uses tree-based interconnects for data distribution and reduction which is similar to our reconfigurable adder tree. T2S [74] is a framework to generate high-performance systolic arrays for dense tensor operations. ExTensor [32] and Tensaurus [73] are sparse tensor accelerators targeting MTTKRP, TTM, SpMV, and SpMM operations. OuterSPACE [61] and SIGMA [69] are SpMM accelerators. OuterSPACE applies OP dataflow. SIGMA proposes a flexible systolic array for different matrix size and a collection network for partial sum accumulation. The collection network decodes the sparse tensor in bitmap format, which shows higher hardware overhead compared with using the BCSR format. SMASH [35] proposed a hierarchical bitmap format and a bitmap management unit for decoding in the CPU platform.

VII. Conclusion

In this article, we propose Morphling, a reconfigurable architecture for efficiently executing both dense and sparse tensor operations. We first propose a flexible tensor execution model which consists of three steps, including tensor vectorization, vector computation, and output reduction. The computation step features three types of parallel patterns that are many-to-one, one-to-many, and one-to-one. The dense and sparse operations differ in the implementation of these patterns. To cooperated with the execution model, we proposed tiled-BCSR format that packs nonzeros into tiles. The architecture of Morphling features a reconfigurable PE array with switches for data communication. The PE can be dynamically configured to support flexible hardware dataflow and enable different types of data reuse. Morphling is synthesized in 28 nm TSMC library with 8.62 mm^2 area at 600-MHz frequency and achieves 13.4X, 677.7X, and 44.7X energy efficiency over Xilinx ZC706 FPGA, Intel i7-9700K CPU, NVIDIA TitanX GPU.

References

[1] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. Symp. Oper. Syst. Design Implement.*, 2016, pp. 265–283.

[2] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *J. Mach. Learn. Res.*, vol. 15, pp. 2773–2832, Jan. 2014.

[3] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proc. Design Autom. Conf.*, 2012, pp. 1212–1221.

[4] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–11.

[5] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proc. Int. Conf. Compiler Construct.*, 2010, pp. 283–303.

[6] J. Bennett and S. Lanning, "The Netflix prize," in *Proc. KDD Cup workshop*, 2007, pp. 35–35.

[7] U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Comput.*, vol. 40, nos. 5–6, pp. 47–58, 2014.

[8] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proc. 21st Annu. Symp. Parallel. Algorithms Architect.*, 2009, pp. 233–244.

[9] B. H. Calhoun, J. F. Ryan, S. Khanna, M. Putic, and J. Lach, "Flexible circuits and architectures for ultralow power," *Proc. IEEE*, vol. 98, no. 2, pp. 267–282, Feb. 2010.

[10] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. Int. Symp. Field Program. Gate Arrays*, 2014, pp. 151–160.

[11] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ACM Sigplan Notices*, 2014, pp. 269–284.

[12] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[13] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Trans. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.

[14] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proc. Int. Symp. Microarchitect.*, 2014, pp. 609–622.

[15] D. C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture design of reconfigurable pipelined datapaths," in *Proc. Conf. Adv. Res. VLSI*, 1999, pp. 23–40.

[16] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitect.*, 2019, pp. 924–939.

[17] B. De Sutter, P. Raghavan, and A. Lambrechts, "Coarse-grained reconfigurable array architectures," in *Handbook of Signal Processing Systems*. Cham, Switzerland: Springer, 2019, pp. 427–472.

[18] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ACM SIGARCH Comput. Archit. News*, 2015, pp. 92–104.

[19] D. M. Dunlavy, T. G. Kolda, and W. P. Kegelmeyer, "Multilinear algebra for analyzing data with multiple linkages," in *Graph Algorithms in the Language of Linear Algebra*. New Delhi, India: SIAM 2011, pp. 85–114.

[20] C. Ebeling, "The general RaPiD architecture description," Dept. Comput. Sci. Eng., Univ. Washington, Seattle, WA, USA, CSE Tech., Rep. UW-CSE-02–06-02, 2002.

[21] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, Jun. 2008.

[22] P. Freyd, "Algebra valued functors in general and tensor products in particular," in *Colloquium Mathematicum*, vol. 14, no. 1, pp. 89–106, 1966.

[23] V. Govindaraju *et al.*, "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep./Oct. 2012.

[24] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, 1978.

[25] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. Int. Symp. Field Program. Gate Arrays*, 2017, pp. 75–84.

[26] S. Han *et al.*, "EIE: efficient inference engine on compressed deep neural network," in *Proc. Int. Symp. Comput. Archit.*, 2016, pp. 243–254.

[27] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. ICLR*, 2016, *arXiv:1510.00149*.

[28] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.

[29] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proc. Asia South Pacific Design Autom. Conf.*, 2001, pp. 564–570.

[30] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proc. Conf. Design Autom. Test Europe*, 2001, pp. 642–649.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[32] K. Hegde *et al.*, "Extensor: An accelerator for sparse tensor algebra," in *Proc. Int. Symp. Microarchit.*, 2019, pp. 319–333.

[33] L. Jia, Z. Luo, L. Lu, and Y. Liang, "Tensorlib: A spatial accelerator generation framework for tensor algebra," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, 2021, pp. 865–870.

[34] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.

[35] K. Kanellopoulos *et al.*, "SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proc. Int. Symp. Microarchit.*, 2019, pp. 600–614.

[36] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCUBE: A CGRA with reconfigurable single-cycle nhop interconnect," in *Proc. Design Autom. Conf.*, 2017, pp. 1–6.

[37] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, 2009.

[38] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[39] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," in *Foundations and Trends in Electronic Design Automation*. Delft, The Netherlands: Now Publ., 2008.

[40] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proc. Int. Symp. Microarchit.*, 2019, pp. 754–768.

[41] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proc. ACM SIGPLAN Notices*, vol. 53, 2018, pp. 461–475.

[42] Y. Liang *et al.*, "An efficient hardware design for accelerating sparse CNNs with NAS-based models," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Mar. 17, 2021, doi: 10.1109/TCAD.2021.3066563.

[43] Y. Liang, L. Lu, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 4, pp. 857–870, Apr. 2020.

[44] Y. Liang, L. Lu, and J. Xie, "OMNI: A framework for integrating hardware and software optimizations for sparse CNNs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 8, pp. 1648–1661, Aug. 2021.

[45] *I. S. Library*. 2019. [Online]. Available: http://isl.gforge.inria.fr/

[46] D. Liu *et al.*, "Pudiannao: A polyvalent machine learning accelerator," in *Proc. ACM SIGARCH Comput. Archit. News*, vol.43, 2015, pp. 369–381.

[47] L. Liu *et al.*, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surveys*, vol. 52, no. 6, pp. 1–39, 2020.

[48] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *Proc. ACM SIGARCH Comput. Archit. News*, 2016, pp. 393–405.

[49] L. Lu *et al.*, "Tenet: A framework for modeling tensor dataflow based on relation-centric notation," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 720–733.

[50] L. Lu *et al.*, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *Proc. MICRO 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 977–991.

[51] L. Lu and Y. Liang, "SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs," in *Proc. Design Autom. Conf.*, 2018, p. 135.

[52] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2017, pp. 101–108.

[53] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. Int. Symp. Field Program. Gate Arrays*, 2017, pp. 45–54.

[54] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proc. Conf. Recommender Syst.*, 2013, pp. 165–172.

[55] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2003, pp. 61–70.

[56] *NELL-2*. [Online]. Available: http://frostt.io/tensors/

[57] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 45, 2017, pp. 416–429.

[58] *CuBLAS*, NVIDIA, Santa Clara, CA, USA, 2019. [Online]. Available: https://developer.nvidia.com/cublas

[59] *CuDNN*, NVIDIA, Santa Clara, CA, USA, 2019. [Online]. Available: https://developer.nvidia.com/cudnn

[60] *CuSPARSE*, NVIDIA, Santa Clara, CA, USA, 2019. [Online]. Available: https://developer.nvidia.com/cusparse

[61] S. Pal *et al.*, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 724–736.

[62] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ACM SIGARCH Comput. Archit. News*, 2017, pp. 27–40.

[63] Y. Park, H. Park, and S. Mahlke, "CGRA express: Accelerating execution using dynamic operation fusion," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2009, pp. 271–280.

[64] A. Pedram, R. A. Van De Geijn, and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Trans. Comput.*, vol. 61, no. 12, pp. 1724–1736, Dec. 2012.

[65] K. K. Poon, S. J. E. Wilton, and A. Yan, "A detailed power model for field-programmable gate arrays," *ACM Trans. Design Autom. Electron. Syst.*, vol. 10, no. 2, pp. 279–302, 2005.

[66] R. Prabhakar *et al.*, "Plasticine: A reconfigurable architecture for parallel patterns," in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 389–402.

[67] PyTorch. 2019. [Online]. Available: https://pytorch.org/

[68] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 24–35.

[69] E. Qin *et al.*, "SIGMA: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 58–70.

[70] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector RDA for sparsity," in *Proc. MICRO*, 2021, pp. 1029–1035.

[71] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proc. Workshop Irregular Appl. Archit. Algorithms*, 2015, pp. 1–7.

[72] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2020, pp. 766–780.

[73] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *Proc. HPCA*, 2020, pp. 689–702

[74] N. Srivastava *et al.*, "T2S-tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *Proc. Int. Symp. Field-Program. Custom Comput. Mach.*, 2019, pp. 181–189.

[75] W.-H. Steeb and T. K. Shi, *Matrix Calculus and Kronecker Product With Applications and C++ Programs*. Singapore: World Sci., 1997.

[76] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[77] M. Vilim, A. Rucker, and K. Olukotun, "Aurochs: An architecture for dataflow threads," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 402–415.

[78] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: accelerating machine learning from relational data," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 309–321.

[79] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proc. 2nd ACM Workshop Online Social Netw.*, 2009, pp. 37–42.

[80] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*, 2019, pp. 1–6.

[81] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "TGPA: tile-grained pipeline architecture for low latency cnn inference," in *Proc. Int. Conf. Comput. Aided Design*, 2018, pp. 1–8.

[82] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. Design Autom. Conf.*, 2017, pp. 1–6.

[83] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *Proc. Int. Conf. Embedded Comput. Syst. Archit. Model. Simulat.*, 2016, pp. 235–244.

[84] Q. Xiao, L. Lu, J. Xie, and Y. Liang, "FCNNLib: An efficient and flexible convolution algorithm library on fpgas," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

[85] Q. Xiao, S. Zheng, B. Wu, X. Pengcheng, X. Qian, and Y. Liang, "HASCO: Towards agile hardware and software co-design for tensor computation," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 1055–1068.

[86] *Xilinx*, San Jose, CA, USA, 2019. [Online]. Available: https://www.xilinx.com/

[87] S. Yin *et al.*, "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications," in *Proc. Symp. VLSI Circuits*, 2017, pp. C26–C27.

[88] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, pp. 687–701.

[89] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. Int. Symp. Microarchit.*, 2016, pp. 1–12.

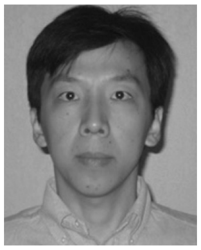
[90] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2020, pp. 261–274.

[91] Y. Zhao *et al.*, "Cambricon-F: machine learning computers with fractal von Neumann architecture," in *Proc. Int. Symp. Comput. Archit.*, 2019, pp. 788–801.

[92] X. Zhou *et al.*, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. Int. Symp. Microarchit.*, 2018, pp. 15–28.

Liqiang Lu received the B.S. degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2017, where he is currently pursuing the Ph.D. degree with the School of EECS.

His research focuses on accelerator design and architecture optimization for deep learning applications, hardware-software co-design to improve hardware efficiency, analysis framework for spatial architecture, and dataflow.



Yun (Eric) Liang (Senior Member, IEEE) received the Ph.D degree in computer science from National University of Singapore, Singapore, in 2010.

He is an Associate Professor (tenure) with the School of EECS, Peking University, Beijing, China. He has authored over 90 scientific publications in premier international journals and conferences in related domains. His research interests include computer architecture, compiler, electronic design automation, and embedded system.

Dr. Liang research has been recognized by best paper awards at FCCM 2011 and ICCAD 2017 and best paper nominations at PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008. He serves as an Associate Editor for *ACM Transactions in Embedded Computing Systems*, *ACM Transactions on Reconfigurable Technology and Systems* (TRETS), and *Embedded System Letters*. He also serves in the program committees in the premier conferences in the related domain, including MICRO, DAC, HPCA, FPGA, ICCAD, FCCM, and ICS.