Automatic Generation of Spatial Accelerator for Tensor Algebra

Liancheng Jia, Zizhang Luo, Liqiang Lu, Yun Liang, *Senior Member, IEEE*,

***Abstract*—Tensor algebra finds applications in various domains including machine learning applications, data analytics and others. Spatial hardware accelerators are widely used to boost the performance of tensor algebra applications. It has a complex hardware architecture and rich design space. Prior approaches based on manual implementation lead to low programming productivity, making it hard to explore the large design space. In this paper, we propose Tensorlib, a framework for generating spatial hardware accelerators for tensor algebra applications. Tensorlib is motivated by the observation that, tensor dataflows can be expressed with linear transformations, and they share common hardware modules which can be reused across different designs. Tensorlib first uses Space-Time Transformation to explore different dataflows, which can compactly represent the hardware dataflow using a transformation matrix. Next, we identify the common structures of different dataflows and build parameterized hardware module templates. Our generation framework can select the needed hardware modules for each dataflow, connect the modules using a specified interconnection pattern, and automatically generate the complete hardware accelerator design. Tensorlib remarkably improves the productivity for the development and optimization of spatial hardware architecture, providing a rich design space with trade-offs in performance, area, and power. Experiments show that Tensorlib can automatically generate hardware designs with different dataflows for a variety of tensor algebra programs. Tensorlib can achieve 318 MHz frequency and 786 GFLOP/s throughput for matrix multiplication kernel on Xilinx VU9P FPGA, which outperforms the state-of-the-art generators.**

I. Introduction

Tensor algebra has been successfully applied to a broad range of compute-intensive applications such as machine learning, data analytics and scientific computation. Tensor algebra features various dimensions, sizes and computation patterns. For example, the 2-D convolution is an important kernel in deep learning applications [26]. It involves a 4-D weight and 3-D input and requires to accumulate the partial sums in four dimensions [26]. Depthwise convolution [8] is an alternative to normal convolution with a smaller workload but different computation pattern. MTTKRP is a widely used tensor operation for tensor factorization in recommendation systems, which takes one 3-D tensor and two 2-D matrices as inputs and generates a matrix. Different accelerator architectures for tensor algebra have been widely deployed on GPU, FPGA and ASIC platforms [5], [6], [23], [30]. For FPGA and ASIC devices, the most commonly adopted hardware architecture is spatial hardware accelerators [6], [51], [53], [54].

The spatial accelerator designs follow a multi-level hierarchical architecture [7], [33]. As shown in Figure 1 (a), spatial accelerators usually consist of an array of homogeneous processing elements (PEs), an on-chip network that connects PEs together, a shared scratchpad buffer, and a system controller. The array of PEs can provide large parallelism, and the connection between PEs can exploit different types of data reuse. While most spatial accelerators adopt a similar hierarchical architecture, the detailed implementation of each design can vary a lot with different design parameters.

Hardware dataflow plays an important role in the behavior of spatial accelerators as it determines how the tensor is computed and communicated between PEs. Initially, dataflow is categorized by specifying the tensors that are reused temporarily inside each PE. For example, Wei et al. [51] uses output stationary systolic array dataflow because the output tensor elements stay inside PE during execution. Similarly, [23] uses weight stationary dataflow since weight value is reused temporally and [6] uses row stationary dataflow.

The complex structure of spatial accelerators leads to large design space and low productivity. It usually requires more than a year to manually design a high-performance hardware accelerator [44]. To reduce the development cycles, High-Level Synthesis (HLS) tools have been used for accelerator designs, which support hardware generation with software-style programming [9], [51]. Some recent works also design Domain-Specific Languages (DSLs) to represent the dataflow and hardware architecture of spatial accelerators [28], [29], [33], [47]. The user can use DSL to express the architecture and the DSL compiler generates hardware code in HLS. However, the auto-generated HLS code is unreadable and hard to optimize, resulting in lower performance compared with manually-optimized implementations.

Recent advances in hardware programming introduce highly parameterized and modular design principles based on object-oriented language. Chisel [1] is a hardware construction language based on Scala and PyMTL [22] is a new hardware language based on Python. The high-level hardware language can be equally expressive as Verilog which supports cycle-level RTL description, but they also support modern programming language features such as functional and object-oriented programming. The high-level programming features enable a large variety of hardware instances to be generated with the same code which defines a set of hardware module templates, which is extremely useful for the design of spatial accelerators

Liancheng Jia, Zizhang Luo, Yun Liang are with Center for Energy-efficient Computing and Applications, Peking University, Beijing, 100871 China. Liqiang Lu is with the college of computer science and technology, Zhejiang University, China. Yun Liang is the corresponding author. (e-mail: {jlc,semiwaker,ericlyun}@pku.edu.cn, liqianglu@zju.edu.cn).

with a variety of dataflows and other configurable parameters.

In this paper, we propose Tensorlib, a framework for generating spatial accelerators for tensor algebra programs. We use Space-Time Transformation (STT) [2] as a means of expression to represent the hardware dataflow. The space-time transformation applies linear transformations on a tensor program to generate the hardware-level scheduling information of the program. STT maps loop instances to hardware execution spatially (coordinates in the PE array) and temporally (timestamp of execution). It can cover the complete design space of spatial dataflows. By inferring the pattern of data reuse using STT, we can represent traditional tensor dataflows used in spatial accelerators including unicast, systolic, multicast, stationary, etc. In addition, Tensorlib also supports the combination of two dataflows on different directions, which becomes a 2-D interconnection pattern.

The accelerator generator needs to support a variety of hardware dataflows, which are different in scheduling and interconnection patterns. However, we observe that different hardware dataflows can share common hardware modules at each level of the hierarchical structure. In other words, a new dataflow can be derived from existing dataflow by only modifying certain parts. Based on these findings, we develop a few basic hardware component templates for each level of hierarchy using Chisel [1], facilitating hardware reuse of different dataflows. When a dataflow is specified by STT, Tensorlib can automatically select the templates and connect them together. Some recent HLS frameworks also apply STT to build the hardware [9], [19], [21], [29], [50], but they are limited in both scalability and performance. Their proposals mainly target systolic architectures which do not cover the complete space of dataflows such as multicast and reduction tree. Instead, Tensorlib supports systolic array, multicast array, reduction tree and the combination of different interconnection topologies. Second, existing frameworks use complex DSL primitives to express tensor programs and hardware dataflows, making it hard to express and optimize. We overcome these challenges by representing the tensor dataflow only based on STT transformation matrix. It effectively simplifies the representation of hardware dataflows, and can be applied to a wide range of tensor programs. Finally, the hardware performance of existing frameworks often cannot match the manual optimized design. In Tensorlib, we perform logical level optimization on the Chisel hardware code template, and the experiments show that our generated hardware code can outperform most existing works.

A preliminary version of this paper was presented in [20], which proposed the Tensorlib framework with basic dataflows (systolic, stationary, multicast, etc). In this paper, we extended the previous version by adding the support for more complex dataflows and analyzing the complete design space of spatial accelerators. Specifically, we support the combination of two basic dataflows, which constructs a 2-D dataflow interconnection. We also build a flexible interconnection template that can be instantiated into hardware dataflow interconnections with different dimensions and shapes. The 2-D dataflow interconnection can be used in many tensor algebra applications including Conv-2D and MTTKRP. We also added more experiments to show the performance variance of different dataflows, and evaluated the performance improvement with physical floorplanning optimizations on FPGAs. Overall, This paper makes the following contributions,

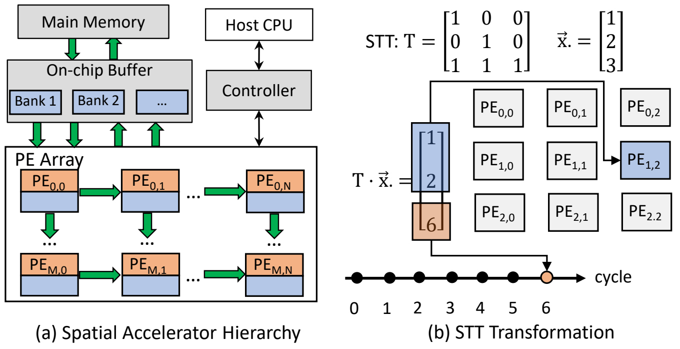

Main Memory
Host CPU
On-chip Buffer
Bank 1
Bank 2
...
Controller
PE Array
$PE_{0,0}$
$PE_{0,1}$
$PE_{0,N}$
$PE_{M,0}$
$PE_{M,1}$
$PE_{M,N}$
(a) Spatial Accelerator Hierarchy
STT: $T = \begin{bmatrix}1&0&0\\0&1&0\\1&1&1\end{bmatrix}$
$\vec{x}. = \begin{bmatrix}1\\2\\3\end{bmatrix}$
$T \cdot \vec{x}. = \begin{bmatrix}1\\2\\6\end{bmatrix}$
$PE_{0,0}$ $PE_{0,1}$ $PE_{0,2}$
$PE_{1,0}$ $PE_{1,1}$ $PE_{1,2}$
$PE_{2,0}$ $PE_{2,1}$ $PE_{2,2}$
cycle
0 1 2 3 4 5 6
(b) STT Transformation

Fig. 1: Spatial accelerator and Space-Time Transformation

- We propose Tensorlib, a framework to automatically generate various hardware dataflow implementation of spatial accelerators for tensor algebra applications. It features rich design space, high productivity, and good performance.
- We develop a formal representation of spatial hardware dataflows using space-time transformation which can cover a comprehensive design space of dataflows.
- Compared with the previous work, we extend the support of 2-D dataflow interconnections, analyze the design space supported by Tensorlib, and evaluate the performance of different dataflows and floorplanning solutions.

Experiment shows that Tensorlib is able to generate a large number of dataflow architectures for various tensor applications. Tensorlib achieves 318 MHz frequency on FPGA for matrix multiplication benchmark with physical optimization, which outperforms most state-of-the-art accelerator generators. Tensorlib also supports more dataflows and applications, which remarkably improves both performance and generality. The source code of Tensorlib is available at https://www.github.com/pku-liang/tensorlib.

II. Background

This section presents the background of spatial hardware accelerator about the structure of PEs, PE array interconnection and on-chip memory. We also introduce the space-time transformation to show its relationship with spatial accelerators.

A. *Spatial Hardware Accelerators*

Spatial accelerators are widely used for accelerating tensor algebra such as convolution, matrix multiplication, tensor contraction and decomposition, etc. Most accelerator designs [6], [12], [41], [46], [51] follow the hierarchical architecture in Figure 1. A typical spatial architecture consists of a 2-D array of processing elements (PEs), an on-chip network

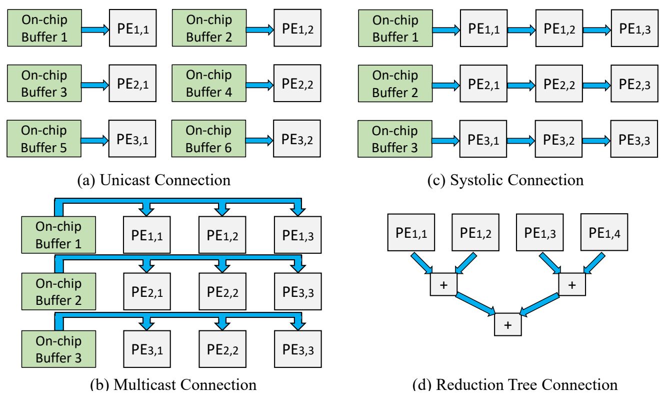


Fig. 2: Dataflows of Spatial Accelerators

that connects PEs together, a shared scratchpad buffer and a system controller. While most spatial accelerators adopt the same hierarchical architecture, the actual implementation of each design can vary a lot. Next, we will show the design choices for each part of the spatial architecture.

PE structure. The PE is composed of a computation kernel module that performs the calculation, and other internal modules that control data movement. The PE computation kernel architecture is determined by the tensor algebra formula, and is often implemented with specialized IPs, such as 1-D convolution [6], vectorized multiplication-accumulation (MAC) [16], [23], [51] or some other computation kernels. The PE internal modules control the data movement inside PE. For example, data needs to be delayed for one cycle in the PE for systolic dataflow, and data needs to be stored in the PE for the entire execution stage in stationary dataflow.

PE array interconnection. The interconnection decides the data communication patterns between PEs and the on-chip buffer. Figure 2 shows four typical PE-array interconnection patterns: unicast, systolic, multicast and reduction tree. In unicast connection, each PE directly connects to the on-chip buffer and data cannot be reused between PEs. In systolic connection, only the first PE connects to the on-chip buffer and other PEs connect together one by one. In multicast connection, all PEs receive data from the on-chip buffer independently. Finally in the reduction tree, partial results from different PEs are reduced with a tree structure to generate result. The tensor algorithm usually involves multiple tensors. Each tensor can have a different interconnection topology, making a large design space for the PE array interconnection.
On-chip memory and controller. The on-chip memory is shared by multiple PEs for low-cost temporal data reuse. The data access pattern depends on the tensor algorithm and the hardware dataflow. Typically on-chip memory is split into banks and each bank supports one load and store request per cycle. Each PE only transfers data with a certain bank to avoid conflicts, and different banks cannot share data directly. Each bank requires an address generator to produce the address for load/store at each cycle. The system controller is implemented with a finite state machine (FSM) that generates control signals for each module so that data can be synchronized properly. For example, the PE requires a signal to control whether to update the stationary data or not, and the on-chip memory modules require signals to synchronize their status.

B. Space-Time Transformation

Space-time transformation (STT) is a linear transformation that maps the tensor algebra to the hardware accelerator from a spatial and temporal perspective. Previous works used STT to analyze systolic arrays [14], [38], [49]. STT can be used for tensor algebra whose computation can be described with nested loops with fixed range, and the dependency between loop iterations must be uniform. Common dense tensor programs such as matrix multiplication and convolution are valid for STT analysis. The PE array can be viewed as a hypercube, and the execution on hardware can be identified as a space-time vector indicating where and when the computation takes place. STT transforms an index in the loop nest to a space-time vector in hardware execution. It can be represented as a matrix multiplication as follows,

$$T\vec{x} = \begin{bmatrix} \vec{p} \\ \vec{t} \end{bmatrix} \tag{1}$$

T is the transformation matrix. $\vec{x}$ is the loop iterator index vector. $\begin{bmatrix} \vec{p} \\ \vec{t} \end{bmatrix}$ is the space-time vector indicating where and when the computation happens in PE array. In particular, space vector $\vec{p}$ means the PE coordinate inside the PE array. Time vector $\vec{t}$ means the time stamp of execution which can also be multi-dimensional. Figure 1 (b) shows an example of STT mapping process. The tensor program is matrix multiplication (GEMM) defined in Equation 2.

$$C[i][j] += A[i][k] \times B[k][j] \tag{2}$$

There are three iterator variables i, j and k. T is the transformation matrix. When $i = 1, j = 2, k = 3$, using equation 1 we can get the result $(1, 2, 6)^T$, where $\vec{p} = (1, 2)^T$ and $t = 6$, which means $A[1][3] \times B[3][2]$ takes place at PE $(1, 2)$ at the sixth cycle. Since a PE can only perform one operation per cycle, matrix T must be a full-ranked matrix so that there is a one-to-one mapping between iterator space and space-time space.

III. Tensorlib Overview

Figure 3 presents the overall flow of Tensorlib framework. The workflow can be divided into two major parts: dataflow generation and hardware implementation generation. For the dataflow generation, Tensorlib uses the tensor algebra program described by a nested loop and an STT matrix (T in Equation (1)) as input. The framework first maps the nested loop into the PE array using STT. Every loop instance is mapped to a spatial and temporal index (space-time vector). Afterwards, by analyzing the recurrence of the same tensor element in different space-time vectors, Tensorlib determines the reuse pattern and dataflow type of each tensor. The computation involves multiple tensors and STT generates

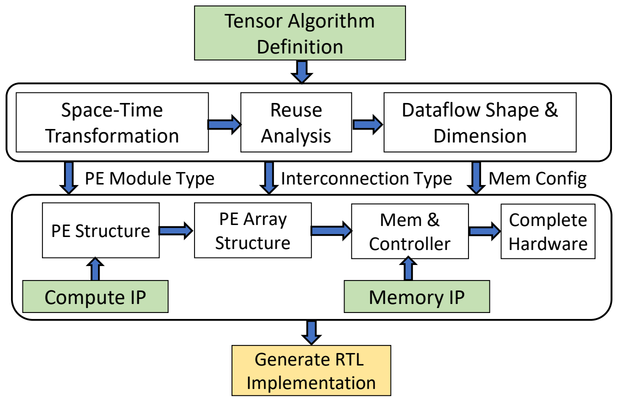


Fig. 3: The overview of Tensorlib

different dataflows for each tensor. Firstly, Tensorlib calculates the dimension of reuse subspace. 0-D, 1-D and 2-D can cover most existing tensor programs. For each type of reuse dimension, Tensorlib further determines the shape of reuse subspace with reuse vectors. An N-D reuse subspace can be expressed by N linearly-independent reuse vectors. For 0-D, 1-D, and 2-D reuse subspace, there are 1, 3, and 6 different cases respectively. Details are discussed in Section IV. We also present an analysis of the design space supported by Tensorlib in section V.

The second step is hardware implementation. The framework generates the 3-level hierarchy of spatial accelerators in a bottom-up manner. Tensorlib pre-defines hardware module templates for PE internal modules, PE-PE interconnection modules, memory modules and PE-memory interconnection modules. Tensorlib first uses the dataflow type of each tensor to select the internal modules of PE and connect them with the computation IP to form the PE structure. Next, it connects the PEs together with a flexible dataflow interconnection template to build the PE array. Finally, it generates the memory modules with access patterns and external memory IPs, and also creates the controller which provides control signals for both PE and memory ports. The generated modules are connected together to form the final accelerator architecture. Details are discussed in section VI.

IV. Dataflow Generation

To represent a dataflow, it is critical to capture the communication behavior of tensor data during the hardware execution. In this section, we first introduce how to generate dataflow with space-time transformation, and then categorize the type of dataflow based on dimension number and shape.

A. Data Reuse Analysis with Space-time Transformation

For an operand in the tensor algorithm, the tensor index $\vec{I}$ accessed at loop index $\vec{x}$ can be expressed as Equation 3, where A is the corresponding access matrix of the operand.

$$\vec{I} = A\vec{x} \tag{3}$$

The same tensor element can be accessed by multiple loop indexes, so the mapping between tensor index and loop index can be 1-to-N. By substituting $\vec{x}$ into $T^{-1}\begin{bmatrix}\vec{p}\\ \vec{t}\end{bmatrix}$ with Equation 1, we can find the relationship between space-time index and tensor index, as shown in Equation 4.

$$AT^{-1}\begin{bmatrix}\vec{p}\\ \vec{t}\end{bmatrix} = \vec{I} \tag{4}$$

To analyze the reuse of tensor elements, we need to find the relationship between different space-time indexes that reuse the same tensor index. For any two space-time indexes $\begin{bmatrix}\vec{p}\\ \vec{t}\end{bmatrix}$ and $\begin{bmatrix}\vec{p}\,'\\ \vec{t}\,'\end{bmatrix}$ which satisfies Equation 4, we have:

$$AT^{-1}(\begin{bmatrix}\vec{p}\\ \vec{t}\end{bmatrix} - \begin{bmatrix}\vec{p}\,'\\ \vec{t}\,'\end{bmatrix}) = \vec{0} \tag{5}$$

The difference of two space-time indexes $\begin{bmatrix}\vec{p}\\ \vec{t}\end{bmatrix}$ and $\begin{bmatrix}\vec{p}\,'\\ \vec{t}\,'\end{bmatrix}$ in Equation 5 illustrates the communication direction of tensor element in the spatial accelerator. Based on this, the kernel space of Equation 5 is a subspace composed of all possible reuse directions of space-time indexes that access the same tensor element. The rank of the kernel space is expressed in Equation 6 based on the rank-nullity theorem [3], where N is the loop dimension.

$$rank(Ker) = N - rank(AT^{-1}) \tag{6}$$

For simplicity, we limit the number of time dimensions $\vec{t}$ to one, and only consider the rank of kernel space to be 0, 1 or 2, which covers most tensor computation dataflows in spatial accelerators.

B. Dataflow Identification with Reuse Space

The shape of reuse space is determined by the rank number, and we discuss them respectively. Firstly, if the rank of reuse space is 0, the space is actually a single point, which means each tensor element is only used once in a particular PE at one cycle and never reused.

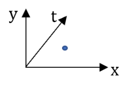
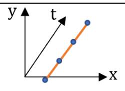
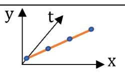
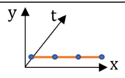
If the rank equals 1, the reuse space is a straight line, and the dataflow is determined by the direction of the line. We can calculate $d\vec{p}$ and dt easily using the basis vector of the kernel space, as shown by Equation 7, where Eig is the eigenvector, E is the identity matrix, $(AT^{-1})^{-}$ is the pseudoinverse of AT^{-1}.

$$\begin{bmatrix}d\vec{p}\\ dt\end{bmatrix} = Eig\left(E - (AT^{-1})^{-}(AT^{-1})\right) \tag{7}$$

There are totally three sub-cases:

- $d\vec{p} = \vec{0}, dt \neq 0$. This case indicates a stationary dataflow. The same tensor element appears in the same PE, but at different time steps.
- $d\vec{p} \neq \vec{0}, dt \neq 0$. This case indicates a systolic dataflow. The tensor element appears in different PEs and at different time steps. dt is usually one, and this corresponds

TABLE I: Dataflow types for 0-D and 1-D space-time reuse space

Subspace Rank	Shape	Space-Time Reuse Space	Tensor Dataflow
0	-		Unicast
1	$d\vec{p}=0, dt \neq 0$		Stationary
	$d\vec{p} \neq 0, dt \neq 0$		Systolic
	$d\vec{p} \neq 0, dt = 0$		Multicast

to systolic dataflow where data are delayed for one cycle before being sent out to other PE.

- $d\vec{p} \neq \vec{0}, dt = 0$. This indicates a multicast dataflow. The tensor element appears in different PEs but at the same time step. The same data must be transformed to different PEs at the same time. If it is an output operand, different partial results are generated simultaneously by different PEs, which requires a reduction tree to generate final results.

Finally, if the rank is 2, the reuse space forms a 2D plane. The plane can be expressed with two linear-independent basis vectors. Each vector belongs to one of the three types in the dim=1 case. For 2D PE array, there are 6 cases of combinations to select two dataflows from three, which are listed below.

- Multicast-Multicast. The tensor element is broadcasted to every PEs in the 2D array at the same time.
- Multicast-Systolic. The tensor element is first multicasted to one dimension of the PE array, and then transfers to all PEs in another dimension with systolic dataflow.
- Systolic-Systolic. The tensor element moves in both dimensions of PE array with systolic dataflow.
- Multicast-stationary. The tensor element keeps stationary during execution, and all PEs in one dimension also share the same data.
- Systolic-stationary. The tensor element keeps stationary during execution, and all PEs in one dimension also share the same data. Although the data transfer is implemented with systolic manner, the data appearance in PE array is the same as the previous case.
- Stationary-Stationary. This case is invalid because there is only one stationary reuse vector. The two vectors cannot be stationary at the same time because they must be linear-independent.

Actually, the systolic-stationary dataflow and multicast-stationary dataflow are equivalent. For multicast-stationary dataflow, the two basis vectors of the reuse space is $v_{m1} = [dx, dy, 0]$ and $v_{m2} = [0, 0, dt]$. The two basis vectors of systolic-stationary dataflow are $v_{s1} = [dx, dy, dt_1]$ and $v_{s2} = [0, 0, dt_2]$. These two reuse spaces are identical because v_{s1} is a linear expression of v_{m1} and v_{m2}. As a result, we combine these two dataflows into one. There are totally four cases of 2-D reuse space. Figure 4 shows the data reuse pattern in the PE array with the four cases of 2D reuse space. The 4x4 matrix is a PE array, and we show the data appearance in three cycles. The blocks with the same color show the reuse pattern of the same tensor index. Each tensor data has a 2-D reuse plane in the 3-D space-time domain.

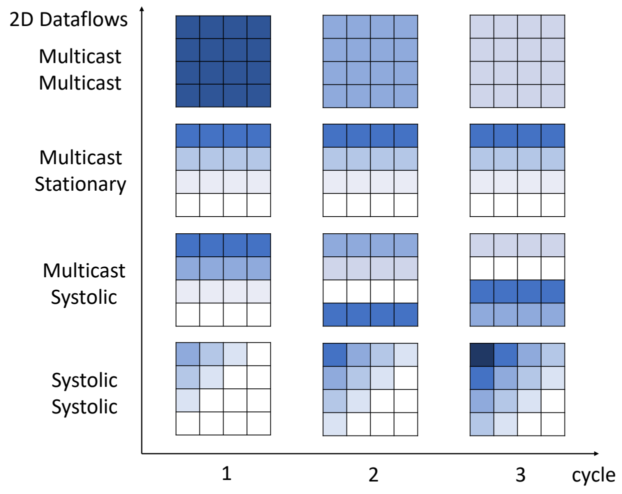


Fig. 4: The data arrangement of PE array with 2D reuse subspace

The dataflow analysis in this section only covers 2D space and 1D time, which can only be mapped with a 3-level loop nest. Typical tensor algebras such as 2D Convolution contain more than three loop nests. For a 2D PE array, we need to select three loops from the loop nest of the tensor program to map to space-time domain. The remaining loops are executed sequentially which doesn't influence PE dataflow. While some accelerators have more than two space loops, such as cluster or multi-core architecture, their interconnections don't support straightforward PE-PE dataflow. For example in a multi-core accelerator, two PEs with the same index but different core cannot transfer data directly. As a result, we don't consider them in our paper.

C. 2D Convolution Example

Now we use the 2D convolution program as an example to explain the dataflow analysis process for a 2D PE array. The computation of 2D convolution is described in Listing 1, which involves three tensors (O, W, I) and six loop iterators (k, c, y, x, p, q). Conv2D program involves six loop iterators, but the space-time analysis only needs three iterators. Therefore, we select three iterators from six, and the other three iterators are removed from the statement. There are $C_6^3 = 20$ cases in total, and we discuss three of them. We use the following STT matrix in Equation 8 and keep it fixed in the discussion.

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{8}$$

```
for(k = 0; k < K; k++)
 for(c = 0; c < C; c++)
  for(y = 0; y < Y; Y++)
   for(x = 0; x < X; x++)
    for(p = 0; p < P; p++)
     for(q = 0; q < Q; q++)
      O[k, y, x] += I[c, y+p, x+q] * W[k, c, p, q]
```

Listing 1: Example Code of Conv2D

Case 1: [K, X, C]. We select $[K, X, C]$ as loop parameters, and keep the other three iterators fixed. Now the statement becomes an ordinary matrix multiplication.

$$O[k, x] += I[c, x] \times W[k, c] \tag{9}$$

We can first find the data access matrix A for the three tensors based on the computation statement in Equation 10. Next, we calculate the kernel space of AT^{-1} for the three tensors. The rank of their kernel spaces is all one, so their reuse directions are 1-D lines. Equation 11 shows the reuse direction (dx, dy, dt) for each tensor.

$$A_O = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, A_I = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, A_W = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} dx_I \\ dy_I \\ dt_I \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} dx_W \\ dy_W \\ dt_W \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} dx_O \\ dy_O \\ dt_O \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{11}$$

With (dx, dy, dt), we can use Table I to find out the dataflow type of each tensor. Tensor I uses horizontal systolic dataflow, tensor W uses vertical systolic dataflow and tensor O uses stationary dataflow. This PE array dataflow corresponds to the traditional output stationary systolic array. The shape of the PE array is $K \times X$, and the number of running cycles is C.

Case 2: [K, X, Q]. In this case, the statement is no longer matrix multiplication. It becomes Equation 12.

$$O[k, x] += I[x + q] \times W[k, q] \tag{12}$$

With the same STT matrix in Equation 8, W and O have the same dataflow as Case 1. However, the dataflow of tensor I becomes different. The rank of tensor I's reuse subspace is 2, so there are two reuse vectors for tensor I, as shown in Equation 13. The first reuse vector is systolic dataflow, while the second one is multicast. As a result, tensor I should apply the multicast-systolic dataflow.

$$\begin{bmatrix} dx_{I1} \\ dy_{I1} \\ dt_{I1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} dx_{I2} \\ dy_{I2} \\ dt_{I2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{13}$$

Case 3: [K, Y, X]. In this case, the statement becomes Equation 14, and we keep the STT matrix unchanged.

$$O[k, y, x] += I[y, x] \times W[k] \tag{14}$$

In this case, there is no reuse opportunity for tensor O and it uses 0-D unicast dataflow. Tensor I has one reuse vector, so it also uses systolic dataflow. Tensor W has two reuse vectors listed in Equation 15, and it uses multicast-stationary dataflow according to STT analysis.

$$\begin{bmatrix} dx_{W1} \\ dy_{W1} \\ dt_{W1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} dx_{W2} \\ dy_{W2} \\ dt_{W2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{15}$$

From the three cases above, we can see that for complex tensor operations, the selection of loops for STT mapping will have a significant impact on the dataflow architecture for the spatial accelerator.

D. Design space analysis

Tensorlib uses two steps to generate a hardware dataflow: loop selection and space-time transformation. Firstly, the user selects three loops from the N-level loop nest to perform the STT analysis. The other loops are iterated temporally which doesn't affect the PE array dataflow. To select three loops from the N-level loop nest, there are $S_{loop} = C_N^3$ different choices.

Next, Tensorlib uses a 3×3 STT matrix to express the concrete hardware dataflow. The STT matrix must be full-ranked so that the mapping between computation instance and space-time vector is one-to-one. To reduce the design space, we do not consider the non-adjacent PE connection patterns by limiting the absolute value of STT matrix elements to 0 or 1. Considering that two dataflows are equivalent if they are rotational or symmetry constant (dataflow B can be derived by dataflow A with rotation or symmetric transformation), the total number of unique dataflows is further reduced. In summary, the number of unique STT matrixes is 105.

The loop selection process is fundamental for the quality of hardware design, since it determines the shape and the reuse opportunity for each tensor. The rank of reuse subspace is determined in this step regardless of STT matrix. When the rank increases, each data element has more reuse times inside PE array without repeated access from SRAM, so the total energy consumption reduces. Furthermore, When the on-chip bandwidth (determined by memory bank numbers) is limited, the lack of PE array reuse (unicast dataflow) can cause delay in the accelerator system. As a result, we should try to avoid the 0-rank reuse space when selecting the loops whenever possible.

Next, the STT design space determines the implementation method for each tensor. Different dataflows can bring different hardware efficiency, and it is also related to PE array size and hardware technology. For example, systolic array dataflow requires the size of time dimension must be larger than space dimension. Otherwise, the pipeline overhead will delay the computation. Besides, systolic array dataflow can achieve better frequency due to the shorter wire distance. Detailed analysis will be discussed in section VI.

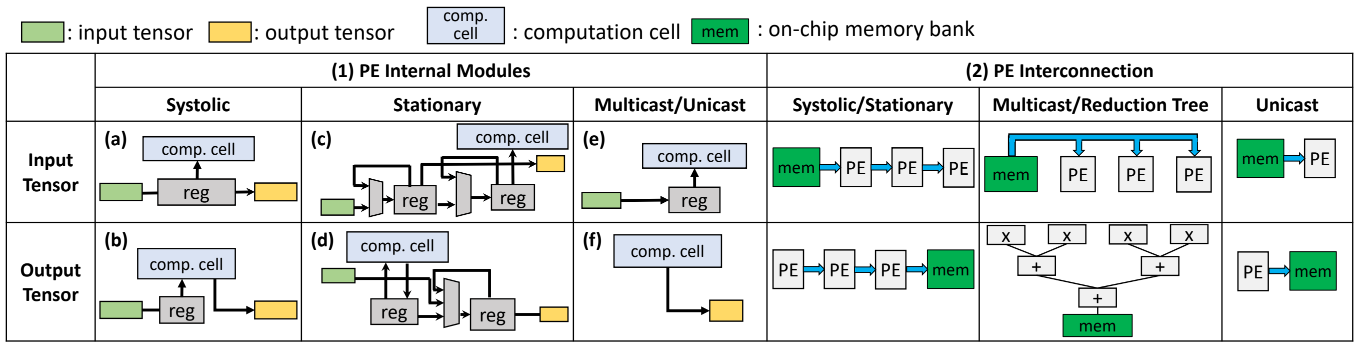
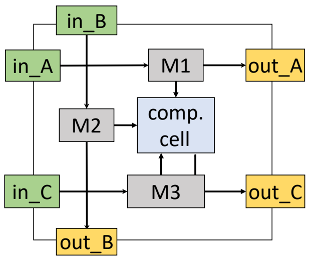


Fig. 5: The PE internal modules and interconnection modules for different dataflows



(a) PE Basic Structure

Dataflow	M1	M2	M3
OS-Systolic	(a)	(a)	(d)
WS-Systolic	(a)	(c)	(b)
Eyeriss	(e)	(c)	(b)
Non-Reusable	(e)	(e)	(f)

(b) Internal Module Selection

Fig. 6: Generating PE structure for different dataflows

V. Hardware Generation

After dataflow generation, the next step is to implement the accelerator architecture with the specific dataflow generated from STT. The architecture of spatial accelerators can be separated into three levels: PE structure, PE array interconnection, on-chip shared buffer and controller. In this section, we discuss how to generate each level of hardware architecture for different dataflows with Tensorlib.

A. PE Generation

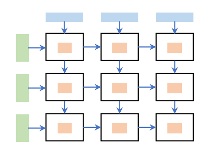
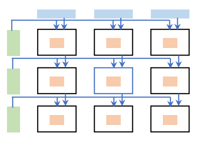
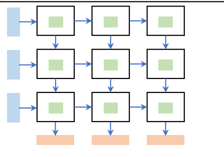
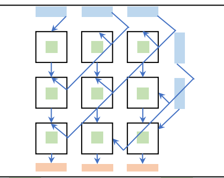
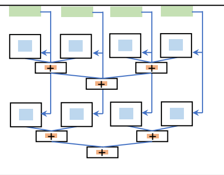
The PE structure consists of the computation kernel and the internal modules that connect the kernel with I/O ports. The computation kernel is a manually implemented IP, which is the same regardless of dataflow variations. The difference in dataflows mainly lies in the internal modules that connect the I/O ports and the computation cells. In Tensorlib, we decouple the I/O interconnection for different operands in the computation. The I/O ports and internal modules of each operand do not connect to each other, and they only connect to the computation kernel, so they can be implemented independently. As shown in Table 1, the dataflow of each tensor inside PE has three types: (a) systolic, (b) stationary and (c) multicast. Each operand can be either input or output of the algorithm. The implementation is different for input and output operands because the output operands need to read results from the computation kernel, so there are 6 conditions in total. In Tensorlib framework, we build the implementation of PE internal modules for each condition, and the circuit diagram is presented in Figure 5.

- Modules (a) and (b) are designed for systolic dataflows. Tensor elements always transfer to their neighboring PEs every cycle. The difference is that the output data is generated from the computation cell and the input data is transferred to the next PE directly.
- Modules (c) and (d) are designed for stationary tensors. Tensor elements stay inside PE during execution, but the data needs to be updated when the execution stage ends. It uses a double-buffer structure to enable the parallelism of computation and data communication. In (d), one register is used to update the result of the current stage, and the other is used to transfer the results of the last stage.
- Modules (e) and (f) are designed for multicast and unicast dataflows, where data are received and passed directly.

Figure 6 shows four examples of PE structure with different dataflow components for 2D Convolution. They share the same basic structure, but PE internal modules M1, M2 and M3 are different. The PE of output-stationary (OS) systolic dataflow contains two modules (a) (systolic, input) and one module (d) (stationary, output). PE for weight-stationary (WS) systolic dataflow contains one (a), one (b) (systolic, output) and one (c) (stationary, input). Row-stationary (RS) dataflow uses direct input module for both input elements and systolic module for output elements. Non-Reusable dataflow uses direct modules for both input and output. The selected components are connected with the PE body and compute cell components to generate the complete PE structure.

B. PE Interconnection Generation

Different dataflows also require different connection topologies between PEs. Figure 5 (2) shows the PE interconnection patterns for 0-D and 1-D dataflows. The systolic and stationary dataflows connect adjacent PEs together. The direction of systolic dataflow interconnection is determined by the reuse vector $(d\vec{p}, dt)$. The output of $PE(x, y)$ connects to $PE(x + dx, y + dy)$ after delaying dt cycle. For multicast input dataflow, the same input data is transferred from the on-chip buffer to PEs in a row directly at the same cycle. The output multicast dataflow is implemented with a reduction tree to perform a reduction on the output of PEs. For unicast dataflow, different PEs are independent and each PE connects to one on-chip memory bank directly.

Dataflow	STT Matrix	PE Connection Pattern	PE Components
(a) Output Stationary Systolic	$\begin{bmatrix}1&0&0\\0&1&0\\1&1&1\end{bmatrix}$		**Input** Systolic **Weight** Systolic **Output** Stationary
(b) Output Stationary Semi-Systolic	$\begin{bmatrix}1&0&0\\0&1&0\\1&0&1\end{bmatrix}$		**Input** Systolic **Weight** Multicast **Output** Stationary
(c) Weight Stationary Systolic	$\begin{bmatrix}0&0&1\\0&1&0\\1&1&1\end{bmatrix}$		**Input** Systolic **Weight** Stationary **Output** Systolic
(d) Eyeriss Row Stationary	$\begin{bmatrix}0&1&0\\0&1&1\\1&0&1\end{bmatrix}$		**Input** Multicast **Weight** Stationary **Output** Multicast
(e) Tree Reduction	$\begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}$		**Input** Stationary **Weight** Multicast **Output** Reduction-Tree

🟦 input 🟩 weight 🟧 output

Fig. 7: Examples of STT matrixes and PE interconnection types for different dataflows for GEMM algorithm

Figure 7 shows five examples of PE interconnection patterns based on different types of dataflow for GEMM (the connection of stationary dataflow isn't shown). In systolic dataflow (a), PEs are connected one by one, and data is transferred to the adjacent PE every cycle in a fixed direction. For multicast input dataflow (b), tensor elements are read from shared on-chip bank and broadcasted to every PEs simultaneously. Eyeriss [6] dataflow uses diagonal connection for input multicast dataflow as shown in (c). Finally, for multicast output dataflow, different PEs generate their partial sums at the same cycle, and they are connected with reduction trees to generate the final results as shown in (d). For 1-D dataflow, each bank connects to one row or column of PEs in the array.

However, when the dataflow becomes 2-D, only one bank is enough to transfer data with all PEs with 2-D interconnection. It requires two steps to build the PE interconnection for 2-D dataflow. Firstly, one reuse vector is selected to build the 1-D interconnection based on the dataflow type and direction. Next, the memory banks in the previous steps are considered as PEs, and the row of ”banks” are connected again with the other reuse vector. The intermediate ”memory banks” are implemented with registers, and finally connected with the real memory bank. As discussed before, the 1-D stationary dataflow can be implemented with systolic interconnection. There are four types of 2-D PE array interconnection topologies(Systolic-Systolic, Systolic-Multicast, Multicast-Systolic, Multicast-Multicast) as shown in Figure 8. Note that (b) and (c) are two implementations of the same dataflow.

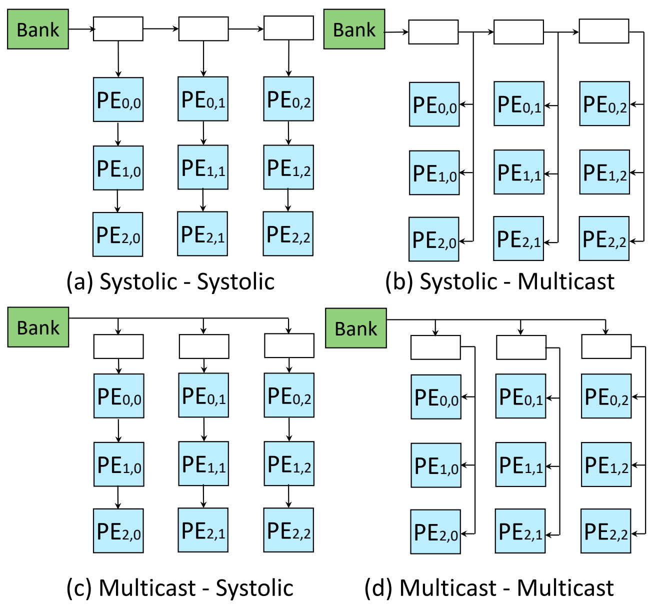

Bank
$PE_{0,0}$ $PE_{0,1}$ $PE_{0,2}$
$PE_{1,0}$ $PE_{1,1}$ $PE_{1,2}$
$PE_{2,0}$ $PE_{2,1}$ $PE_{2,2}$
(a) Systolic - Systolic
(b) Systolic - Multicast
(c) Multicast - Systolic
(d) Multicast - Multicast

Fig. 8: Four types of PE interconnection for 2-D dataflow

C. Memory and Controller Generation

Tensorlib automatically generates the on-chip memory module and connects to the PE array with the 1-D or 2-D interconnection pattern discussed in the previous section. The data inside each memory module is mapped with the tensor index according to the PE index it connects with, and the cycle number that accesses the particular memory address. Equation (4) gives the algorithm to get the tensor data with PE index and cycle number. Tensorlib also builds an address generator for each on-chip memory bank that controls the memory read and write address at every cycle.

The system controller provides the control signal to PEs in order to ensure that the execution of all the PEs are synchronized. The control signal controls the validity of the memory's output port where the PE array receives data, and also the PE's internal components. When the PE array receives valid data, it triggers computation. Otherwise, it keeps idle. For the PE internal components, only the stationary components (Figure 5(c, d)) need control signals to control the stationary tensor. The system controller maintains a state machine and it provides the control signal for all the PEs with stationary internal modules. When the dataflow is specified, the pattern of control signals is also determined.

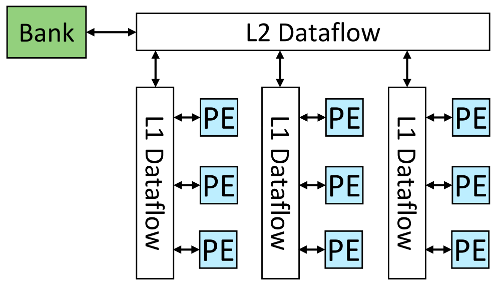


Fig. 9: The 2-D PE array interconnection template

TABLE II: Evaluated Tensor Algebra Programs

Name	Formula
GEMM	$C[m,n] += A[m,k] \times B[n,k]$
Batched-GEMV	$C[m,n] += A[m,k,n] \times B[m,k]$
Conv2D	$C[k,y,x] += A[c,y+p,x+q] \times B[k,c,p,q]$
Depthwise-Conv	$C[k,y,x] += A[k,y+p,x+q] \times B[k,p,q]$
MTTKRP	$D[i,j] += A[i,k,l] \times B[k,j] \times C[l,j]$
TTMc	$D[i,j,k] += A[i,l,m] \times B[l,j] \times C[m,k]$

D. Implementation Details

As discussed in section III, the Tensorlib framework is implemented with a front-end dataflow generator program and a back-end hardware generator. For the front-end dataflow generator, Tensorlib provides a Scala-based DSL for users to write the tensor program and the STT matrix. Users can specify the range of each loop to perform tiling, and also set the bit-width of each tensor operand. Afterwards, the user can call the Tensorlib API to generate the hardware dataflow configurations.

For the back-end part, we utilize Chisel's object-oriented programming feature to develop the hardware templates. They are implemented with Chisel classes that extends the same I/O interface, but their internal structures are different according to the dataflow type. To support the dataflow with different dimensions smoothly, Tensorlib uses a 2-D dataflow interconnection template to connect each memory bank with PEs, as shown in Figure 9. It can be flexibly configured into different interconnection types. Each level of dataflow can be configured into multicast (reduction tree), systolic and unicast interconnections. For 1-D dataflow, one of the two dimensions uses unicast interconnection. For 0-D dataflow, both dimensions use unicast interconnection. The computation kernel inside each PE is implemented with chisel *BlackBox* module so that it can be substituted with user-defined computation IPs. We also write a default integer MAC kernel with flexible bit-width in Chisel. We find that the address generator module often becomes the performance bottleneck for design frequency, since it requires a complex tensor indexing expression. We build a specialized FSM and add more levels of registers to reduce the combinational calculation complexity of the address generator to achieve better frequency. Finally, all the modules are connected in a fixed hierarchical way to build the complete accelerator architecture.

TABLE III: Evaluated Dataflows for Matrix Multiplication

Dataflow	Tensor A	Tensor B	Tensor C
OS-Sys	Systolic	Systolic	Stationary
OS-Half	Systolic	Multicast	Stationary
OS-Mul	Multicast	Multicast	Stationary
WS-Sys	Stationary	Systolic	Systolic
WS-Half	Stationary	Multicast	Systolic
WS-Tree	Stationary	Multicast	Reduction Tree

VI. Evaluation

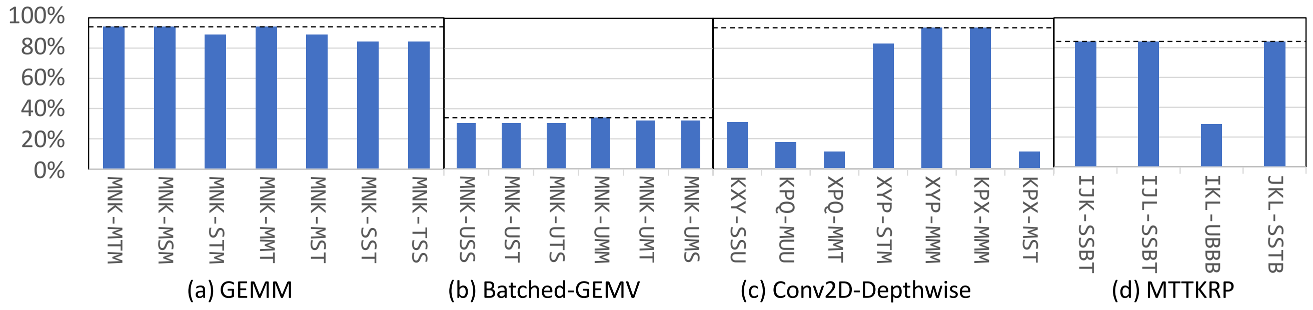
To evaluate the performance of Tensorlib, we first test the cycle-level performance of the generated hardware code with different benchmarks and dataflow. Next, we evaluate the power and area performance of different dataflow designs. We also evaluate the resource utilization and frequency performance on FPGAs, and compare with previous works to show the performance improvement brought by Tensorlib.

The Chisel code generated by Tensorlib is compiled to Verilog and then synthesized on both ASIC and FPGA platforms. For ASIC evaluation, we use Synopsys DC compiler with 55nm UMC 1P8F technology for synthesis, and Synopsys VCS for simulation. For FPGA tests, we set the target device to Xilinx VU9P FPGA with 6840 DSPs and 2160 BRAMs. We use Vivado 2021.2 version for synthesis and implementation. For FPGA floating-point operations, we use Xilinx's Floating-Point IP and integrate it into Chisel implementation as a BlackBox module.

A. Cycle-Level Performance Results

For cycle-level performance simulation, We evaluate six tensor applications: GEMM, Batched-GEMV, Conv2D (2 layers from ResNet), Depthwise-Conv2D, MTTKRP and TTMc. The formula of each tensor algebra is shown in Table II. For each tensor, we use S, T, M, U, B to refer to systolic, stationary, multicast (reduction tree), unicast, 2-D dataflow, respectively. To represent a dataflow using STT, we first select three loop iterators from the loop nest. Therefore, in the following, we name the dataflow of the hardware using the selected loop iterators and the dataflow of each tensor. For example, XPQ-MMT refers to selecting X, P, Q loops to perform STT transformation, and use multicast dataflow for A and B, and Stationary for C, respectively. For Conv2D, XYP-MMT is the dataflow with multicast connection, KCX-SST and KCX-STS are well-known output-stationary and weight-stationary systolic array dataflows [16], [23], [51]. XYP-MST is a combination of systolic and mutlicast dataflow, which is similar as ShiDiannao [12].

We set the size of PE array to 16×16, it runs under 320MHz frequency with 32GB/s on-chip bandwidth between PE array and scratchpad memory. Figure 10 presents the normalized performance of several representative dataflows measured by execution cycles compared with peak performance (full PE array utilization). As shown, different dataflows vary greatly in performance.



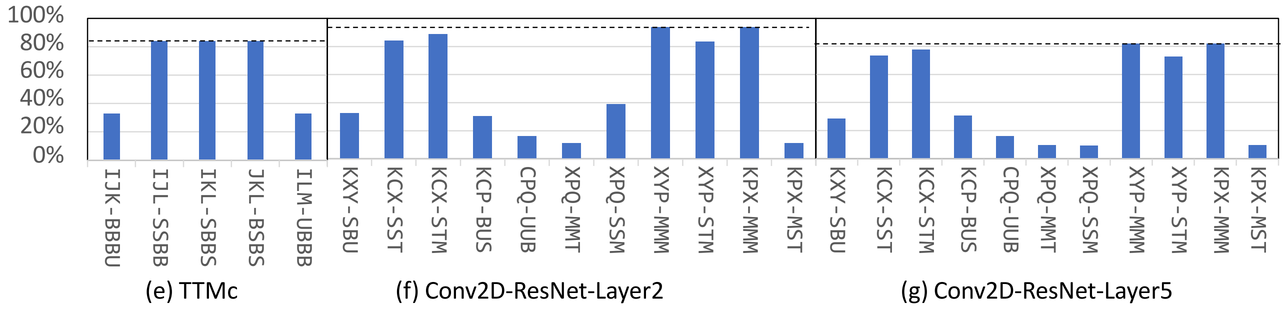


Fig. 10: Normalized performance of different dataflows for each tensor algebra

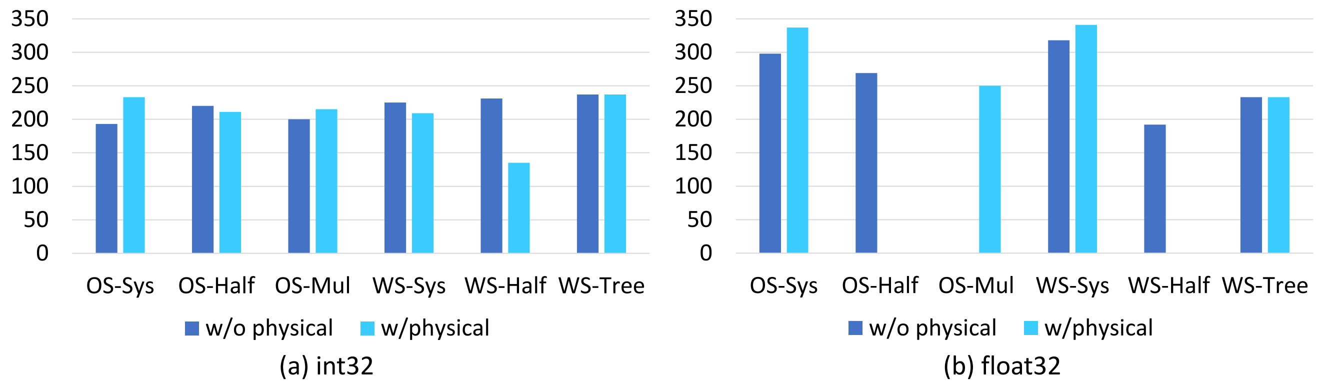


Fig. 11: FPGA frequency results of different dataflow, data type and physical constraints

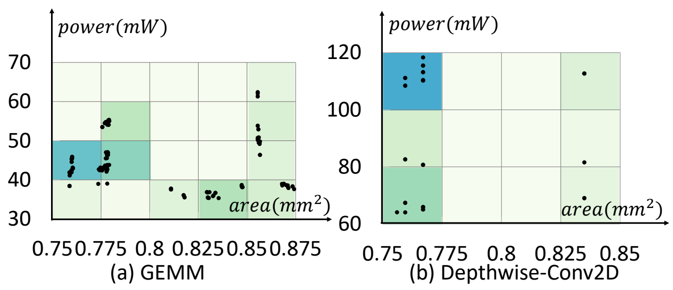


Fig. 12: Power and area result of different dataflow designs

For GEMM benchmark, the multicast dataflow (MTM) uses less cycle numbers than systolic dataflow (STS) because multicast dataflows have a smaller pipeline overhead than the systolic array. But systolic array is preferred in hardware because of the lower interconnection cost and better frequency. For MTTKRP and TTMc, the unicast dataflows (e.g. IKL-UBBB and IJK-BBBU) perform worse than others because unicast dataflows require all PEs to transfer data with on-chip memory simultaneously and bandwidth becomes insufficient. The STT matrix can be adjusted to avoid unicast for better performance. However, Batched-GEMV can only use unicast dataflow because the tensor A is only accessed once and cannot be reused during computation, requiring a large on-chip bandwidth.

As discussed in section IV.C, Conv2D has a lot of loop selection choices, and each of them has different STT matrixes, leading to a large design space. The range of kernel size loop is often relatively small (e.g. Conv2D kernel size P and Q can be 3 or 5 in some layers), leading to low utilization of PEs when mapping to PE array dimension. For example, XYP-SMM and KPX-TMM have 1/16 idle PEs since the range of p is 3 and

TABLE IV: Performance comparison with prior works of GEMM and Conv2D on FPGA Platforms

	[40]	X.wei et al. [51]	AutoSA [50]		Susy [29]		[11]	Tensorlib	
Platform	ARRIA-10	ARRIA-10	Alveo-U250		ARRIA-10		VU9P	VU9P	
Workload	GEMM	Conv2D	GEMM	Conv2D	GEMM	Conv2D	GEMM	GEMM	Conv2D
Data Type	float32	float32	float32	float32	float32	float32	float32	float32	float32
Array Size	10×16	11×14	13×16	16×14	-	10×8	-	12×13	12×13
Vectorization	8	8	8	8	-	16	-	8	8
# of multipliers	1280	1232	1664	1792	1417	1280	-	1248	1248
LUT	-	59%	52%	58%	40%	35%	81%	68%	73%
BRAM	-	45%	41%	30%	32%	30%	80%	51%	51%
FF	-	40%	42%	46%	-	-	46%	50%	50%
DSP	99%	81%	68%	73%	93%	84%	48%	75%	75%
Freq(MHz)	313	271	300	272	202	220	146	318	303
Gop/s	800	667	934	950	547	551	409	786	749

only 15 out of 16 rows of PE are used. The performance of ResNet-Layer5 is even lower because X and Y loops are also small (x=y=7). For some other dataflows, although every PE is assigned with workloads, it becomes idle in some cycles because execution can only start when data reaches the farthest PE from the memory bank. When the execution cycle is small, the communication delay can be larger than computation. For example in the KPX-MST case, 3 elements transfer across the systolic array with 16 cycles, leading to 81% idle cycles during execution. For Conv2D workloads, selecting KCX iterations can deliver better performance for most layers, because it becomes a standard GEMM operation with large loop bounds. However, for Depthwise-Conv, all reduction dimensions are small, so regular Conv2D dataflows cannot be applied to Depthwise-Conv. Results show that KPX-MMM and XYP-MMM dataflow perform better for Depthwise-Conv.

B. ASIC synthesis Evaluation

Here, we evaluate a large design space of dataflows for GEMM and Depthwise-Conv2D in a 16×16 PE array for INT16 datatype. We set the target frequency to 320MHz for ASIC synthesis. Figure 12 gives the area and power performance of different dataflow architectures generated by Tensorlib. Each point in the graph refers to one dataflow, and there are totally 148 points for GEMM and 33 points for depthwise-Conv2D. Experiments show that dataflow choice has a larger impact on power than area consumption. The power variation of GEMM range from 35mW to 63mW, which shows 1.8X difference, while the area has only 1.16X difference. Compared with other dataflows, dataflow with two multicast input (MMT, MMS) consumes more power because of the fan-out structure. However, reduction tree output dataflow doesn't cost more power, although it uses more hardware resources. Dataflows with stationary tensor consume more area and power because of the control signals for stationary data.

C. FPGA Performance Evaluation

Different accelerator designs have a large impact on the FPGA performance. We evaluate three major design choices in our experiment: data type, dataflow and physical design. We discuss the effect of each design choice respectively. We use the GEMM operation as the benchmark, and set the PE array size to 12×13. The SIMD vectorization degree is 8. We evaluate INT32 and FP32 datatype, and test six different dataflows listed in Table III. ”Tree” stands for reduction tree dataflow.

TABLE V: FPGA Resource utilization comparison on different datatype and dataflows

Benchmark	LUT	DSP	FF	BRAM
Float w/o tree	77%	73%	50%	53%
Float w/ tree	94%	90%	61%	53%
Int w/o tree	28%	54%	20%	53%
Int w/ tree	34%	72%	26%	53%

1) Resource utilization results: We compare the FPGA resource utilization rates of the generated accelerators with floating-point and integer datatype, and show the result in Table V. We find that different data types lead to different resource usages. Floating-point consumes 2.5x to 3x LUT and FF resources compared with integer datatype, but only 1.3x DSP resources. This is because each FP32 MAC unit consumes four DSPs while each INT32 MAC unit uses three. Since both datatype has 32-bit data width, they consume the same BRAM resources. Dataflow with reduction trees also increases DSP usage, since the addition units are located in each reduction module.

2) Frequency performance results: We synthesize each of the designs on Vivado tools to obtain the frequency performance. The results are shown in Figure 11 (a) (b) for INT32 and FP32 datatypes. For each design, we also try physical floorplanning optimizations by adding placement constraints to force each PE to reside in one specific SLR, in order to prevent the nets with too long distance. Xilinx UV9P has three SLRs, and the size of PE array is 12×13. So we divide the PE array with three 4×13 matrices, and set each of them into one SLR. The result are shown in 11 (c) and (d).

From the results we find that dataflow variance has a different impact on INT32 and FP32 datatype. INT32 is not sensitive to dataflow and physical constraints. Most of them have 200 to 230MHz frequency. The reason is that INT32 datatype uses fewer FPGA resources, and the critical path exists inside the computation kernel, so dataflow and physical constraints cannot optimize the critical path. For FP32 tests, the systolic dataflows perform much better than non-systolic dataflows. Also, physical constraints can effectively boost frequency for systolic dataflows but has no improvements on non-systolic dataflows. However, some test cases failed to route due to the congestion problem, as shown in OS-Mul, OS-Half and WS-Half tests in Figure 11 (b) and (d). The best performance for FP32 is 318 MHz without physical constraints and 341 MHz with physical constraints. Since systolic dataflow has only adjacent PE-PE interconnection, it is more sensitive to the distance between PEs. With physical constraints to reduce the distance between adjacent PEs, the performance becomes better. On the contrary, the physical constraints can even hurt the performance in some other cases, because the interconnection of non-systolic dataflows cannot benefit from the PE matrix layout, and Vivado's original layout strategies are also affected.

3) Comparison with other works: We also compare Tensorlib with other FPGA spatial accelerators including AutoSA [50], Susy [29], Wei *et al.* [51], Johannes *et al.* [11] and Srivatsan *et al.* [40]. The result is shown in Table IV. Susy [29] and [51] can generate the hardware implementation of systolic arrays with a polyhedral model or space-time transformation. However, their designs are limited in performance and supported algorithms. Both Susy and Wei's work only support systolic array dataflow, and they fail to generate hardware for algorithms that don't fit well in systolic architecture, such as the GEMV program. AutoSA supports more tensor programs including MTTKRP, depthwise convolution, but it also only supports systolic architecture. In contrast, Tensorlib supports more dataflow variants including multicast and reduction tree structures. Our synthesis result achieves 318 MHz frequency and 786 Gop/s throughput, which outperforms all existing works in frequency. The throughput is less than AutoSA since AutoSA uses a larger FPGA with more DSPs, but Tensorlib achieves a higher DSP resource utilization (75% vs 68-73%).

AutoSA uses Autobridge [17] to improve the frequency by assigning each hardware module to one particular FPGA slot to minimize the slot-crossing cost. Inspired by this, we also manually optimize the physical placement using physical constraints in Vivado toolchain. This improves the frequency of our design to 341 MHz on VU9P, which is 13.6% higher than AutoSA.

VII. Related Works

Dataflow Design of Spatial Accelerators. Modern spatial accelerators explore different dataflows optimization techniques for efficient execution of tensor applications. [57] [32] [53] presented dataflow accelerator with different dataflows for DNN on FPGAs. Systolic arrays have been a popular spatial architecture for tensor algebra acceleration [13], and has been adopted in Google's TPU [23]. Eyeriss [6] differentiate different dataflows according to the data reuse strategy, and proposed a row-stationary dataflow for convolution. ShiDianNao [12] designs a special architecture that uses systolic dataflow for input and multicast for weight. NVDLA [41] and Flexflow [35] presented dataflow for DNN processing and Caffeine [56] presented a full-system evaluation. Morphling [34] and OMNI [31] designs a reconfigurable dataflow for both dense and sparse tensor computation, but it cannot support efficient spatial data reuse. [24] [15] presents efficient dataflows by using 3D-stacked memory in the accelerator. Most of the related works target the DNN domain. Guo *et al.* [18] studied the performance impact of broadcasting dataflow on FPGA device.

Performance Analysis framework of Spatial Accelerators. Many previous works develop systematic frameworks or performance models to analyze and evaluate the performance of spatial dataflow accelerators. Ma *et al.* [36] optimize loop order, tiling and unrolling to minimize the memory access on each storage levels. dMazeRunner and Timeloop [10], [42] develop cost models to explore the optimization space of multi-level tiling for perfectly nested loops. These three frameworks mainly focus on the efficient mapping of tensor algebra on a fixed spatial architecture. Intersteallar [55] considers the dataflow variation on PE array level including systolic array and reduction tree architecture by defining extra notations with Halide [43], but it only covers a small subset of PE array architecture compared with Tensorlib. MAESTRO [27] analyzes spatial and temporal reuse of tensors by mapping loops to different dimensions of PE array execution. Tenet [33] uses relation-centric notations to represent dataflow and build performance models, which can be equivalent to the STT-based notation, but it only considers 1-D dataflow. Scale-sim [45] builds a cycle-accurate simulator specially for systolic array architecture. It considers the impact of dataflow, array shape and memory hierarchy, but it is limited to CNN algorithm and systolic architecture.

Hardware Generation Frameworks of Spatial Accelerators. Some works are proposed to automate the generation of spatial accelerator with HLS tools. Wei *et al.* [51] and PolySA [9] develop generation tools only for systolic array architecture with polyhedral transformation for FPGAs. Gemmini [16] is a systolic array generator that supports system-level parameters but can only generate two dataflows. MAGNet [48] proposes a hardware generator with several stationary dataflows with flexible PE interconnections. DSAGEN [52] presents an accelerator generation tool by using an abstract description of hardware architecture. Spatial [25] is a programming language based on Chisel for hardware accelerator generation. Halide [43] first presents the idea of DSLs, and following works develop their own DSLs for hardware synthesis [47] [39] [28] [37] [48]. Machine learning frameworks such as TVM and Flextensor [4], [58] are also adding the support for hardware code synthesis. In contrast, Tensorlib can thoroughly explore the dataflow design space using space-time transformation and build parameterized and reusable modules for efficient hardware generation.

VIII. Conclusion

In this paper, we propose TensorLib, a framework for generating spatial accelerators. First, we present a formal expression of spatial accelerator dataflow with Space-Time Transformation (STT). Tensorlib uses STT to analyze the tensor reuse behavior to generate the dataflow type and communication direction for each tensor. We build reusable hardware module templates for each dataflow and automatically select hardware modules to construct PE structure, PE interconnection, on-chip memory and controller for a complete spatial accelerator. Experiment results show that Tensorlib can generate various accelerators with different dataflows on FPGA and ASIC platforms and achieve similar or better performance than prior HLS and HDL-based designs.

ACKNOWLEDGMENT

This work is supported in part by National Natural Science Foundation of China (NSFC) under grant No.62090021.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *DAC*, 2012.

[2] Donald G. Baltus and Jonathan Allen. Efficient exploration of nonuniform space-time transformations for optimal systolic array synthesis. In *ASAP*, 1993.

[3] Sudipto Banerjee and Anindya Roy. *Linear algebra and matrix analysis for statistics*, volume 181. Crc Press Boca Raton, 2014.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Geoff Voelker Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, Al Davis Olivier Temam, and Sarita V. Adve. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[6] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*, 2016.

[7] Yu-Hsin Chen, Tien-Ju Yang, Joel S. Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 2019.

[8] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *CVPR*, 2017.

[9] Jason Cong and Jie Wang. Polysa: polyhedral-based systolic array auto-compilation. In *ICCAD*, 2018.

[10] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Trans. Embedded Comput. Syst.*, 2019.

[11] Johannes de Fine Licht, Grzegorz Kwasniewski, and Lesley Shannon Torsten Hoefler. Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis. In *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, 2020.

[12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ISCA*, 2015.

[13] Clément Farabet, Berin Martini, B. Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *CVPR*, 2011.

[14] José AB Fortes, K-S Fu, and Benjamin W Wah. Systematic design approaches for algorithmically specified systolic arrays. In *Computer architecture*. Elsevier Science Inc., 1988.

[15] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*, 2017.

[16] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert J. Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *CoRR*, 2019.

[17] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die fpgas. In *FPGA*, 2021.

[18] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency. In *DAC*, 2020.

[19] Liancheng Jia, Liqiang Lu, Xuechao Wei, and Yun Liang. Generating systolic array accelerators with reusable blocks. *IEEE Micro*, 2020.

[20] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. Tensorlib: A spatial accelerator generation framework for tensor algebra. In *DAC*, 2021.

[21] Liancheng Jia, Yuyue Wang, Jingwen Leng, and Yun Liang. Ems: Efficient memory subsystem synthesis for spatial accelerators. In *DAC*, 2022.

[22] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 2020.

[23] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[24] Duckhwan Kim, Jaeha Kung, Sek M. Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *ISCA*, 2016.

[25] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: a language and compiler for application accelerators. In *PLDI*, 2018.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[27] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *MICRO*, 2019.

[28] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *FPGA*, 2019.

[29] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher J. Hughes, and Pradeep Dubey. Susy: A programming model for productive construction of high-performance systolic arrays on fpgas. In *ICCAD*, 2020.

[30] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *FPL*, 2016.

[31] Yun Liang, Liqiang Lu, and Jiaming Xie. Omni: A framework for integrating hardware and software optimizations for sparse cnns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[32] Yun Liang, Qingcheng Xiao, Liqiang Lu, and Jiaming Xie. Fcnnlib: A flexible convolution algorithm library for deep learning on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[33] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. TENET: A framework for modeling tensor dataflow based on relation-centric notation. In *ISCA*, 2021.

[34] Liqiang Lu and Yun Liang. Morphling: A reconfigurable architecture for tensor computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[35] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*, 2017.

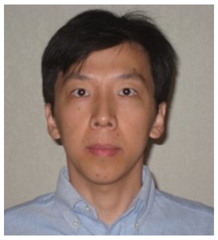
[36] Yufei Ma, Yu Cao, Sarma B. K. Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *FPGA*, 2017.

[37] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Hipa^{cc}: A domain-specific language and compiler for image processing. *IEEE Trans. Parallel Distrib. Syst.*, 2016.
[38] Willard L Miranker and Andrew Winkler. Spacetime representations of computational structures. *Computing*, 1984.
[39] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. VTA: an open hardware-software stack for deep learning. *CoRR*, 2018.
[40] Duncan J. M. Moss, Krishnan Srivatsan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit K. Mishra, Debbie Marr, Suchit Subhaschandra, and Philip Heng Wai Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *FPGA*, 2018.
[41] Nvidia. Nvdla deep learning accelerator. 2019.
[42] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel S. Emer. Timeloop: A systematic approach to DNN accelerator evaluation. In *ISPASS*, 2019.
[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Saman P. Amarasinghe, and Cormac Flanagan. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.
[44] Hongbo Rong. Programmatic control of a compiler for generating high-performance spatial hardware. *CoRR*, 2017.
[45] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883*, 2018.
[46] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, 2016.
[47] Nitish Kumar Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David H. Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, Geoff Lowney, Adam Herr, Christopher J. Hughes, Timothy G. Mattson, and Pradeep Dubey. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *FCCM*, 2019.
[48] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Ross Pinckney, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J. Dally, Joel S. Emer, Stephen W. Keckler, and Brucek Khailany. Magnet: A modular accelerator generator for neural networks. In *ICCAD*, 2019.
[49] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Process.*, 1991.
[50] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on FPGA. In *FPGA*, 2021.
[51] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput CNN inference on fpgas. In *DAC*, 2017.
[52] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: synthesizing programmable spatial accelerators. In *ISCA*, 2020.
[53] Qingcheng Xiao, Liqiang Lu, Jiaming Xie, and Yun Liang. Fcnnlib: An efficient and flexible convolution algorithm library on fpgas. In *DAC*, 2020.
[54] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. HASCO: towards agile hardware and software co-design for tensor computation. In *ISCA*, 2021.
[55] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, Mark Horowitz, Luis Ceze, and Karin Strauss. Interstellar: Using halide's scheduling language to analyze DNN accelerators. In *ASPLOS*, 2020.
[56] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In *ICCAD*, 2016.
[57] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.
[58] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *ASPLOS*, 2020.

Liancheng Jia received the B.S. degree in computer science from Peking University. Liancheng Jia is working toward the Ph.D. degree with the Center for Energy-Efficient Computing and Application, Peking University. His current research interests include high-performance computation in GPU and embedded systems. His research focuses on the automatic generation and optimization of hardware accelerators.

Zizhang Luo is currently is working toward the Ph.D. degree with the Center for Energy-Efficient Computing and Application, Peking University. His research interest focuses on algorithms and architectures for machine learning systems.

Liqiang Lu is an associate professor (ZJU100 Young Professor) in the college of computer science and technology, Zhejiang University, China. He obtained his Ph.D. degree from the school of computer science, Peking University, China in 2022. His research focuses on quantum computing, deep learning accelerator design, spatial architecture optimization. He has authored over 20 scientific publications in premier international journals and conferences in related domains, including ISCA, MICRO, FCCM, TCAD, DAC, etc.



Yun (Eric) Liang is currently an Associate Professor (with tenure) at Peking University. His research interest is at the hardware-software interface with work spanning electronic design automation (EDA), hardware and software co-design, and computer architecture. He has authored over 100 scientific publications in the leading international journals and conferences. His research has been recognized with two Best Paper Awards (FCCM 2011 and ICCAD 2017), six Best Paper Award Nominations (PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, CODES+ISSS 2008). He currently serves as Associate Editor of the ACM Transactions on Embedded Computing Systems (TECS) and ACM Transactions on Reconfigurable Technology and Systems (TRETS). He is the program chair of International Conference on Field Programmable Technology (FPT) 2022 and International Conference on Application-specific Systems, Architecture and Processors (ASAP) 2019 and the subcommittee chair of Asia South Pacific Design Automation Conference (ASPDAC) 2014. He also serves in the program committees in the premier conferences including DAC, ICCAD, DATE, ASPDAC, FPGA, FCCM, HPCA, MICRO, PACT, CGO, ICS, CC, CASES, LCTES, ASAP, and ICCD.