Rubick: A Synthesis Framework for Spatial Architectures via Dataflow Decomposition

Zizhang Luo*
School of Integrated Circuits
Peking University
semiwaker@pku.edu.cn

Liqiang Lu*
College of Computer Science and Technology
Zhejiang University
liqianglu@zju.edu.cn

Size Zheng
School of Computer Science
Peking University
zhengsz@pku.edu.cn

Jieming Yin
School of Computer Science
Nanjing University of Posts and Telecommunications
jieming.yin@njupt.edu.cn

Jason Cong
Computer Science Department
University of California at Los Angeles
cong@cs.ucla.edu

Jianwei Yin
College of Computer Science and Technology
Zhejiang University
zjuyjw@cs.zju.edu.cn

Yun Liang†
School of Integrated Circuits
Peking University
ericlyun@pku.edu.cn

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247743

***Abstract*—Dataflows are critical for spatial architectures designed for tensor applications. Prior works develop various notations and hardware generation frameworks for dataflows. However, due to the semantic gap between notations and low-level details, analysis based on these notations cannot capture the detailed architectural features between different dataflows, so these works failed to provide architectural optimization and efficient design space exploration (DSE) at the same time.**

We propose Rubick, a synthesis framework for spatial architecture. Rubick decomposes the dataflow into two low-level intermediate representations including access entry and data layout. Access entry specifies how data enter into the PE arrays from memory, while data layout specifies how data are arranged and accessed. Based on this decomposition, Rubick provides efficient DSE and generates optimized hardware. Experiments show that the DSE time is accelerated by up to $1.1 \times 10^5 X$ and performance on FPGA is improved by 13%.

***Index Terms*—Dataflow architectures, Automatic synthesis**

I. Introduction

Spatial architectures play a pivotal role in the acceleration of various tensor applications [5], [7], [10], [12], [19]. A typical spatial architecture features a processing element (PE) array with a scratchpad memory, which exhibits high compute parallelism and energy efficiency. Besides, there are abundant interconnection resources that connect PEs to support different datapaths and enable efficient data reuse. Spatial architecture is a natural fit for tensor applications, whose computation and memory access are highly regular but demand high performance.

Hardware dataflow is the key component when implementing applications onto spatial architectures. Recently, several frameworks have been proposed for dataflow analysis and performance modeling [8], [9], [14], [15], [17], [18], [25]. These existing frameworks model the behaviors of loop instances that intertwine multiple tensors, and model the spatial architecture in its entirety. However, different tensors might have distinct characteristics and behavior (e.g., dimension, size, movement, etc.). It is hard to infer the behavior of each tensor from the high-level notation, including data access patterns and data arrangements. On the other hand, spatial architectures consist of PEs and memoy, which require different low-level architectural features.

Several works have proposed methods to analyze the architectural details and synthesize RTL implementation from high-level representation [10]–[12], [16], [22]–[24]. Tensorlib [10] generates PE arrays by extracting three types of reuse vectors from space-time transform, namely systolic, multicast, and stationary. Tensorlib only exploits dataflow reuse. EMS [12] further analyzes the memory subsystem by exploiting the intra-bank and inter-bank reuse vectors. However, they focus on hardware generation for a specific or a limited range of designs and do not provide a way to efficiently explore the design space. TENET [17] can represent the complete design space of hardware dataflows using relation-centric notation, but lacks an efficient DSE approach. Moreover, since it neglects the architectural implementation details, TENET can not be used for dataflow optimization with specific hardware constraints such as fan-in/fan-out.

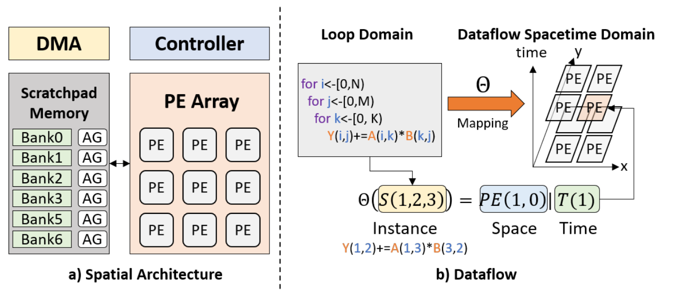
In this paper, we propose Rubick, a synthesis framework for spatial architecture. We analyze the structure of spatial accelerators and propose two new low-level IRs to expose architectural details: *access entry* and *data layout* to describe the computation and memory, respectively. The spatial architecture dataflows can be decomposed into these two IRs and vice versa, which provides opportunities for efficient design space exploration (DSE). For each IR, we form a design sub-space according to their characteristics, to eliminate illegal or inefficient designs, which dramatically reduce the total space. Finally, we propose an end-to-end synthesis flow based on these two IRs, which can automatically explore the design space and generate the spatial hardware implemented in Register-Transfer-Level (RTL).

The contributions of this paper are:

- We propose two new IRs access entry and data layout to bridge the gap between dataflow and hardware. The new IRs can express low-level details and can be composed to express high-level information.
- We form the complete design space of dataflows according to the new IRs. The decomposition in space improved DSE efficiency.
- We present an end-to-end synthesize flow of spatial architecture.

Our experiments demonstrate that Rubick can reduce 82.4% wire resources with only 2.7% latency increase by optimizing access entry IR. Rubick also accelerates the DSE time of dataflows by 1.6×10^3X - 1.1×10^5X, saving the time from several days to minutes compared to TENET [17]. The implementation on FPGAs shows Rubick can find a better dataflow for 2D-CONV which improves the performance by 13% compared with TensorLib [10] and EMS [12].

* These authors contributed equally.
† Corresponding Author



```
for i<-[0,N)
  for j<-[0,M)
    for k<-[0, K)
      Y(i,j)+=A(i,k)*B(k,j)
```

Fig. 1. (a) Spatial architecture overview. (b) Dataflow mapping illustration and an example.

II. Background

A. Tensor Basics

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.

Iteration domain and loop instance. Given a loop nest with one statement, its iteration domain D_S is the set that contains all the loop instances. Each instance S is labeled with a loop iterator $\vec{n}$ consisting of loop variables i, j, $\cdots$.

$$D_S = \{S(\vec{n}) \mid \vec{n} = (i, j, \cdots)\}$$

Tensor domain. The tensor domain is the set of all the elements in the tensor. The dimension of the tensor domain is the same as its order. The tensor domain of tensor A is denoted using $\vec{n'}$ consisting of tensor indices.

$$D_A = \left\{A(\vec{n'})\right\}$$

Access function. Given a loop instance, the access function returns the tensor elements used by this instance, which can be regarded as a mapping from iteration domain to tensor domain.

$$\mathcal{A}_{D_S \to (D_A, D_B, \ldots)} = \{S(\vec{n}) \to (A(\vec{n'_A}), B(\vec{n'_B}), \cdots)\} \quad (1)$$

For example, the instance, tensor domain, access function of GEMM is written as follows.

$$S(\vec{n}): \quad Y(i,j) += A(i,k) \times B(k,j), \ \ \vec{n} = (i,j,k)$$
$$D_A = \{A(\vec{n'}) \mid \vec{n'} = (i,k)\}$$
$$\mathcal{A}_{D_S \to (D_A, D_B)} = \{S(i,j,k) \to (A(i,k), B(k,j))\}$$

B. Spatial Architecture

Spatial architecture is an accelerator architecture that has the structure shown in Fig. 1 (a). PE array and scratchpad memory are the most important parts of the architecture. In this work, we assume the PE array is a 2-dimensional matrix of PEs, denoted as a space-stamp $PE(\vec{p})$, $\vec{p} = (x, y)$. Each PE contains a simple arithmetic logic unit (ALU) and some data registers. We assume each cycle a PE can operate once, denoted as a time-stamp $T(\vec{t})$, $\vec{t} = (t1, t2, \cdots)$. We use the multi-dimensional time to resemble the loop nest, where t_is are loop variables and $t1$ is the innermost loop, and the sequence is determined by the lexicographical order of time-stamp $T(\vec{t})$. PEs interconnects with each other to allow data reuse between neighbors, which can effectively increase data reuse opportunities and reduce the memory bandwidth requirement. PEs can also access data from scratchpad memory. The scratchpad memory is separated into banks to support the parallel access of PEs. Address generators (AGs) are used to select data from the banks.

C. Spatial Dataflow

The key component of the spatial architecture is the dataflow that determines how a tensor kernel is mapped onto the architecture. The dataflow is a mapping from one loop instance to a space-stamp and time-stamp.

$$\Theta_{D_S \to D_{st}} = \{S(\vec{n}) \to (PE(\vec{p}) \mid T(\vec{t}))\} \quad (2)$$

Dataflow spacetime domain (D_{st}) is the domain that consists of multiple spacetime-stamps (space-stamp and time-stamp), where each spacetime-stamp refers to a PE at a certain cycle. $\Theta_{D_S \to D_{st}}$ assigns a loop instance $S(\vec{n})$ from the iteration domain to a spacetime-stamp from the dataflow spacetime domain. The space-stamp $PE(\vec{p})$ gives the coordinates of PE where the instance will be executed, and the time-stamp $T(\vec{t})$ gives the execution sequence. For example, in Fig. 1 (b), $S(1,2,3) \to (1,0|1)$ means the instance i=1, j=2, k=3 is executed in PE(1,0) at cycle 1.

Tensor movement. A tensor movement is a part of the dataflow, which gives the information of only one tensor at a time. Given a tensor domain for a target tensor A with its index vector $\vec{n'}$, the tensor movement is defined as a mapping from the dataflow spacetime domain to the tensor domain.

$$M_{D_{st} \to D_A} = \{(PE(\vec{p}) \mid T(\vec{t})) \to A(\vec{n'})\} \quad (3)$$

III. Dataflow Decomposition

The main novelty of Rubick is to decompose the dataflow into two low-level intermediate representations (IRs): *access entry* and *data layout*, which are expressive enough for architecture implementation. Another benefit of the decomposition approach is the efficient design space exploration. By defining rigorous architectural constraints for the subspace corresponding to each IR, the combined space can be significantly pruned.

Observing that the PE array and the scratchpad memory are two major components of the spatial architecture, we take the entry points between these two components as a natural separation, and thus a tensor movement can be decomposed into two IRs. We derive these two IRs by defining a new domain called the *entry spacetime domain*, which acts as an intermediate domain.

Definition 1: **Entry spacetime domain** (E_{st}) is defined as the domain that consists of multiple spacetime-stamps $E_{st} = \{(E(\vec{p_e}) \mid T(\vec{t_e}))\}$. The spacetime-stamp refers to an entry port $E(\vec{p_e})$ at a certain cycle $T(\vec{t_e})$, which loads data from memory and sends them to the PE array.
With this new domain, we can bridge the gap between dataflow notation and architecture implementation.

In this section, we first give the formal definition of access entry, data layout, and decomposition (Section III-A). Then, we use an example to illustrate how dataflow decomposition helps with architecture implementation (Section III-B).

A. Dataflow Decomposition

To decouple each tensor behavior from a computational instance, we first decompose it into different tensor movements by applying the access function of each tensor.

$$\Theta_{D_{st} \to D_S} = \left(M_{D_{st} \to D_A} \otimes M_{D_{st} \to D_B}, \ldots\right) \times \mathcal{A}_{(D_A, D_B, \cdots) \to D_S} \quad (4)$$

Here, we choose to use the Cartesian product symbol $\otimes$ because the merged access function maps to the Cartesian space of all tensors. The $\times$ symbol means the chain composition of two mappings. Considering that the output tensor indices of most tensor applications are determined by the indices of input tensors, we only decompose the dataflow into movements of input tensors in this paper. Taking GEMM as an example,

$$\Theta_{D_{st}\to D_S} = \left(M_{D_{st}\to D_A} \otimes M_{D_{st}\to D_B}\right) \times A_{(D_A,D_B)\to D_S}$$

The tensor movement is further decomposed into access entry Ω and data layout L. This helps to decouple the PE array part and memory part from spatial architecture.

Definition 2: **Access entry.** Given a dataflow spacetime domain D_{st} of a dataflow, the access entry is defined as a mapping from D_{st} to the entry spacetime domain E_{st}.

$$\Omega_{D_{st}\to E_{st}} = \{(PE(\vec{p_d}) \mid T(\vec{t_d})) \to (E(\vec{p_e}) \mid T(\vec{t_e}))\} \tag{5}$$

Here, $(PE(\vec{p_d}) \mid T(\vec{t_d}))$ is a dataflow spacetime-stamp that takes place in $PE(\vec{p_d})$ at the time-stamp $T(\vec{t_d})$. The tensor used by this dataflow spacetime-stamp comes from the entry space-stamp $E(\vec{p_e})$ at the entry time-stamp $T(\vec{t_e})$. If two dataflow spacetime-stamps refer to the same entry spacetime-stamp, it means they use the same tensor data.

From an architectural perspective, access entry indicates how to design the on-chip memory. The space-stamp $\vec{p_e}$ tells the dimension of memory banks and their allocation. On the other hand, the time-stamp $\vec{t_e}$ describes the access pattern of tensor data, which further determines the PE interconnection.

Definition 3: **Data layout.** Given an entry spacetime domain E_{st} and tensor A domain D_A, the data layout is defined as a mapping from E_{st} to D_A,

$$L_{E_{st}\to D_A} = \{(E(\vec{p_e}) \mid T(\vec{t_e})) \to A(\vec{n'})\} \tag{6}$$

Mathematically, this intermediate representation maps the indices in the entry spacetime domain to the tensor indices. Therefore, it explicitly depicts which tensor element is used by the entry $E(\vec{p_e})$ at $T(\vec{t_e})$. Here, the term *data layout* is a general definition that not only describes the data arrangement spatially but also the sequence of the tensor accessed to/from entry points. Moreover, the tensor size determines the boundary of each time dimension, which further decides the memory size.

By defining access entry and data layout, the decomposition of tensor movement is formulated as follows.

$$M_{D_{st}\to D_A} = \Omega^A_{D_{st}\to E_{st}} \times L_{E_{st}\to D_A} \tag{7}$$

Taking GEMM as an example, the decomposition formula is written as follows.

$$\begin{aligned}\Theta_{D_{st}\to D_S} = &\left(\Omega^A_{D_{st}\to E_{st}} \times L_{E_{st}\to D_A}\right)\\ \otimes &\left(\Omega^B_{D_{st}\to E_{st}} \times L_{E_{st}\to D_B}\right)\\ \times &\ \mathcal{A}_{(D_A,D_B)\to D_S}\end{aligned} \tag{8}$$

B. Decomposition Example

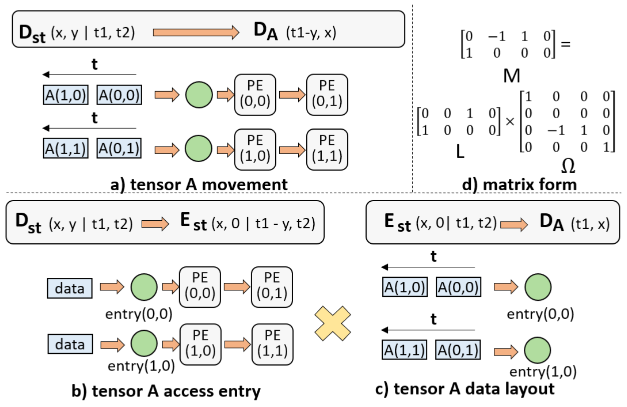
In this subsection, we use GEMM dataflow as an example to illustrate dataflow decomposition. The dataflow is written as

$$\Theta_{D_S\to D_{st}} = \{S(i,j,k) \to PE(k, j\%2) \mid T(i + j\%2, j/2)\}$$

where the matrix size is set to $0 \le i < 2$, $0 \le k < 2$, $0 \le j < 4$. This dataflow involves 2 spatial dimensions (2×2 PE array), and 2 time dimensions (6 cycles in total).

Here, we only demonstrate the decomposition of tensor A. As shown in Fig. 2 (a), according to Equation 4, the tensor movement of A is written as follows.

$$M_{D_{st}\to D_A} = \{PE(x,y) \mid T(t1,t2) \to A(t1-y,x)\}$$



$$\begin{bmatrix}0&-1&1&0\\1&0&0&0\end{bmatrix} = \begin{bmatrix}0&0&1&0\\1&0&0&0\end{bmatrix} \times \begin{bmatrix}1&0&0&0\\0&0&0&0\\0&-1&1&0\\0&0&0&1\end{bmatrix}$$

Fig. 2. GEMM dataflow decomposition example. The data movement (a) of tensor A is decomposed into access entry IR (b) and data layout IR (c). The mathematical process can be written in matrix form (d).

For simplicity, we write the dataflow spacetime-stamp D_{st} in Fig. 2 as $\{(x, y \mid t1, t2)\}$.

Then, as shown in Fig. 2 (b) and (c), we formulate the access entry and data layout of input tensor A according to Equation 7.

$$\begin{aligned}\Omega^A_{D_{st}\to E_{st}} &= \{(x,y \mid t1,t2) \to (x,0 \mid t1-y,t2)\}\\ L_{E_{st}\to D_A} &= \{(E(x,0) \mid T(t1,t2)) \to A(t1,x)\}\end{aligned}$$

Identifying that the entry space-stamp is a 1D-vector (the second dimension of the entry space-stamp is 0), we know that there is only one memory bank of tensor A for PEs in the same row. On the other hand, this IR maps $(x, y \mid t1, t2)$ and $(x, y+1 \mid t1+1, t2)$ in D_{st} to the same entry $(x, 0 \mid t1-y, t2)$, indicating that elements of tensor A horizontally traverse across the PE array (along the y-axis). Thus, it requires building interconnections between adjacent PEs in the same row when designing the PE interconnection. The data layout of tensor A further indicates that each memory bank should store a column of tensor A, where all $A(t1, x)$ are accessed at entry $(x, 0)$.

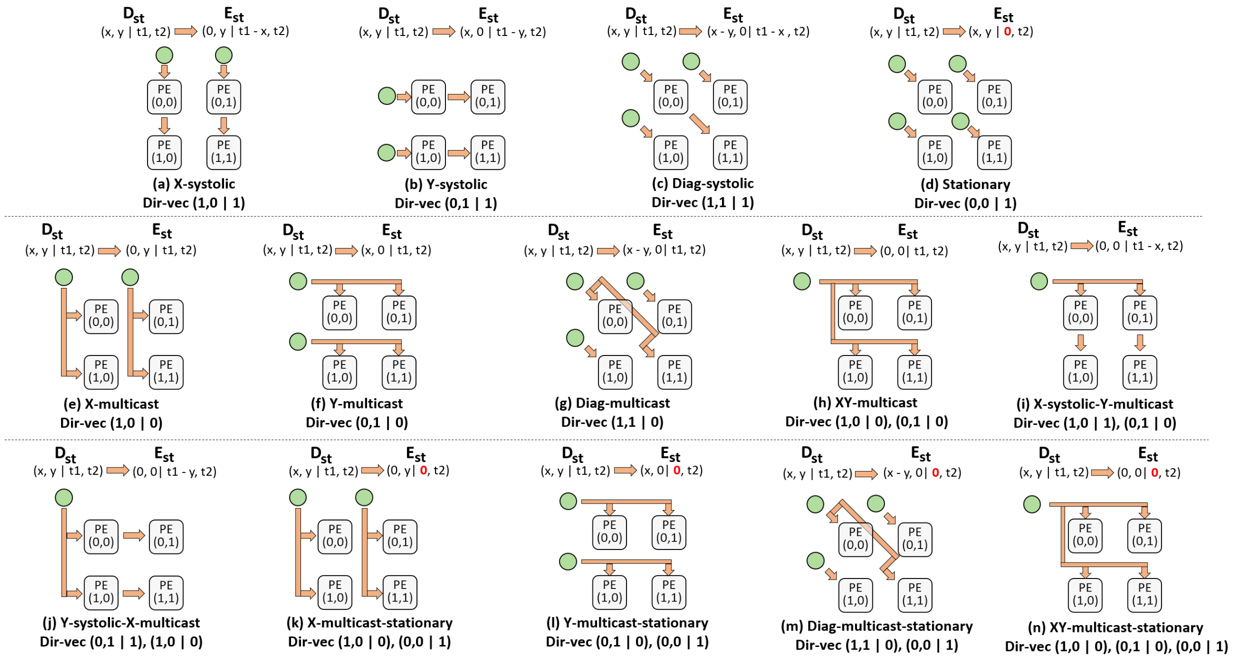
The tensor movement is in fact the chain product of the access entry and the data layout, which is equivalent to the matrix multiplication equation in Fig. 2 (d). The tensor movement in the figure can be separated into two parts. The access entry is the right part and the data layout is the left part. Note that, the access entry IR only tells there is a data access from entry to PE and its access direction. By composing it with data layout IR, we can exactly figure out what exactly this data is. For example, the data used by entry $(0, 0 \mid 1, 0)$ is A(1,0).

IV. Dataflow Design Space

For a given dataflow, we can specify one of them and calculate another according to Equation 7. Or, we can specify both to compose the complete dataflow. Therefore, we can form the access entry space and data layout space separately. The access entry space is formed as a linear space that consists of multiple linear combinations of access direction vectors (Section IV-A). The data layout space enumerates all possible linear transformations that map spacetime-stamps to the tensor domain (Section IV-B).

A. Access Entry Space

We assume that data are always accessed linearly, thus, the access entry can be formulated as a linear combination of base vectors. For example, access patterns like $A[ai + j]$ are considered linear while $A[i^2]$ is non-linear and not supported by our model. From an architectural perspective, the base vector equals to direction vector



(a) X-systolic Dir-vec (1,0 | 1)
(b) Y-systolic Dir-vec (0,1 | 1)
(c) Diag-systolic Dir-vec (1,1 | 1)
(d) Stationary Dir-vec (0,0 | 1)
(e) X-multicast Dir-vec (1,0 | 0)
(f) Y-multicast Dir-vec (0,1 | 0)
(g) Diag-multicast Dir-vec (1,1 | 0)
(h) XY-multicast Dir-vec (1,0 | 0), (0,1 | 0)
(i) X-systolic-Y-multicast Dir-vec (1,0 | 1), (0,1 | 0)
(j) Y-systolic-X-multicast Dir-vec (0,1 | 1), (1,0 | 0)
(k) X-multicast-stationary Dir-vec (1,0 | 0), (0,0 | 1)
(l) Y-multicast-stationary Dir-vec (0,1 | 0), (0,0 | 1)
(m) Diag-multicast-stationary Dir-vec (1,1 | 0), (0,0 | 1)
(n) XY-multicast-stationary Dir-vec (1,0 | 0), (0,1 | 0), (0,0 | 1)

Fig. 3. Input access entry space on 2D-PE array. The space is formulated as tensor access direction vectors.

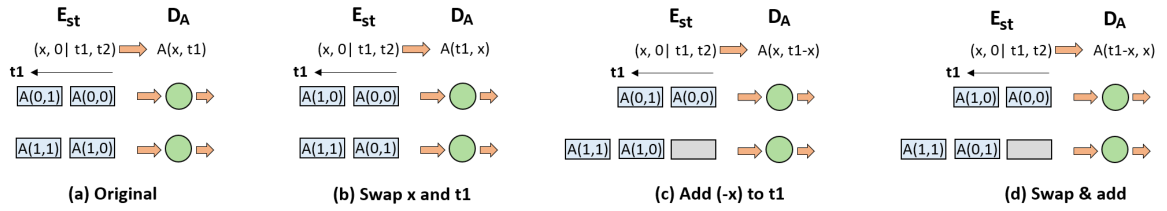


Fig. 4. Data layout space on 2D-PE array. The space is formulated as linear matrix transformation.

(dir-vec) $\vec{r}$ that indicates the direction of how tensor elements are accessed across spatial dimension and time dimension. For a given access entry, its direction vectors $\vec{r}$ all satisfy $M_{D_{st}\to D_A}(\vec{r}) = 0$. Inversely, we can derive a unique access entry from a set of direction vectors. According to the aforementioned assumption, the reuse direction vector is a triple $(x, y \mid t)$. In this manner, there are 7 basic direction vectors in total.

X-systolic: $(1, 0 \mid 1)$ **Y-systolic:** $(0, 1 \mid 1)$ **Stationary:** $(0, 0 \mid 1)$
X-multicast: $(1, 0 \mid 0)$ **Y-multicast:** $(0, 1 \mid 0)$
Diag-systolic: $(1, 1 \mid 1)$ **Diag-multicast:** $(1, 1 \mid 0)$

As these vectors form a 3D space at most, the number of direction vectors for a specific access entry is up to 3. The number of all possible direction vector combinations is $C_7^1 + C_7^2 + C_7^3 = 63$. After removing the repeated linear space and symmetric linear space, there are only 14 access entry types. Fig. 3 lists all of them on a 2D-PE array. Fig. 3 (a)-(c) are systolic patterns with horizontal, vertical and diagonal (slope = 1) data transfer. In Fig. 3 (d), the first dimension of time-stamp is 0, representing each PE keeps the tensor element stationary for a while. Fig. 3 (e)-(g) are multicast networks where entries spatially distribute like a 1D-vector. The last six access entries in Fig. 3 (i)-(n) are hybrid patterns.

Note that Fig. 3 only depicts the cases of input access entry. By reversing the access direction, it can also be applied to output access entry. For example, multicast access entry means the partial sums are generated simultaneously, while systolic access entry means the partial sums are generated in continuous cycles.

B. Data Layout Space

Data layout space depends on both application (tensor domain) and architecture (entry spacetime domain). We apply linear matrix transformation when forming its space. Mathematically, there are only three basic transformations: 1) swap two rows, 2) add one row to another, and 3) multiply a row by a factor. The third one only occurs in quasi-affine transformation if tiling is needed. The tensor access behavior mainly depends on the first two transformations. Fig. 4 depicts how the linear transformation affects the data layout. Fig. 4 (b) swaps the order of spatial dimension x and the innermost time dimension $t1$ when mapping the indices in E_{st} to the indices in D_A. Compared to Fig. 4 (a), it acts like a transposition when tensor A is a matrix. In Fig. 4 (c), we add $(-x)$ to $t1$ in E_{st} and map it to D_A, leading to data skewing.

V. Synthesis Flow

Rubick can efficently explore the design space of spatial hardware and determines the optimal hardware design for a given tensor application under certain constraints. The flow is shown in Fig. 5.

Input: Rubick accepts a tensor expression and some hardware constraints like buffer capacity and resources as inputs. The access entry space is accepted as an optional input in our flow to support different target platforms, which may support different a subset of the space, and may have a different implementation. For example,

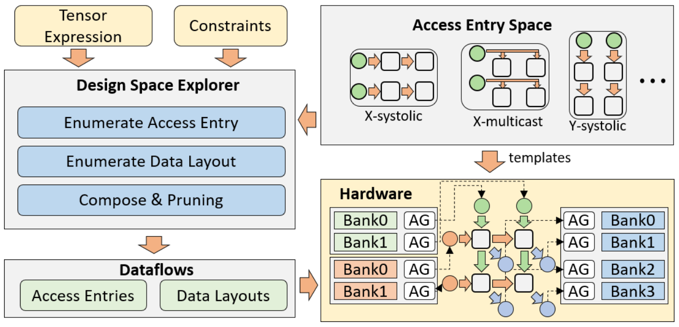


Fig. 5. Rubick Synthesis flow

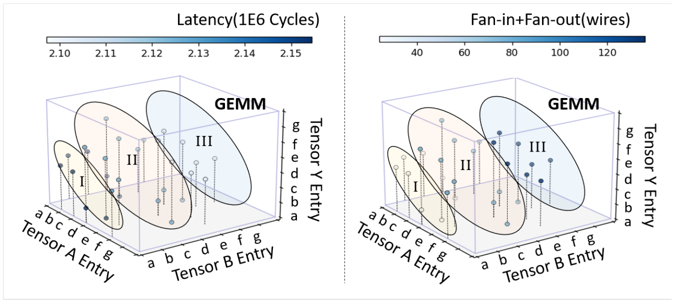


Fig. 6. Using Rubick access entry IRs to explore hardware design. a-g means the access entry type in Figure 3.

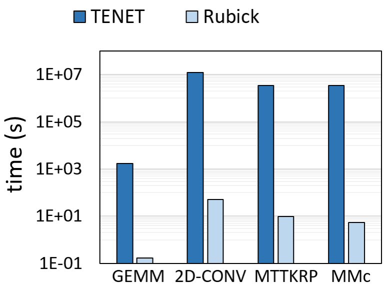


Fig. 7. Exploration-efficiency improved by Rubick.

multicast access entries for input tensors may suffer bad frequency on FPGAs. The user may specify the desired subspace and replace the template if needed.

DSE: Rubick automatically explores the dataflows within the design space. We separately enumerate the choices of access entry and data layout for each tensor, which dramatically reduces the total space. Other design parameters like the height and width of the PE array can be also searched optionally. Some pruning techniques based on the mathematical properties of the two IRs are applied, e.g. legal dataflows should be a one-to-one mapping, time can not be negative, etc. As a branch-and-prune algorithm, the theoretic time complexity of DSE is exponential to the number of loops and tensors, but dataflow decomposition allows early pruning of illegal points so the actual DSE time is acceptable.

Output: Rubick outputs all legal dataflows in the form of two IRs. RTL hardware can be generated according to the dataflows with certain backends. Specifically, the PE array is composed of access entry templates of each tensor, and the scratchpad memory is composed of memory banks and address generators according to data layouts. The templates are written in Chisel [3] for parametric generation.

VI. Experiment

A. Experiment Setup

Benchmarks. We evaluate the following benchmarks.

$$\begin{array}{ll}\textbf{GEMM} & Y(i,j) += A(i,k)B(k,j) \\ \textbf{2D-CONV} & Y(n,k,ox,oy) += A(k,c,rx,ry)B(n,c,ox+rx,oy+ry) \\ \textbf{MMc} & Y(i,j) += A(i,k)B(k,l)C(l,j) \\ \textbf{MTTKRP} & Y(i,j) += A(i,k,l)B(k,j)C(l,j)\end{array} \tag{9}$$

GEMM and 2D-CONV are single kernels, which are widely used in deep learning and scientific computing [1], [2], [13]. Matrix multiplication chain (MMc) is used in the attention mechanism of transformer models [6], [20]. Matricized tensor times Khatri-Rao product (MTTKRP) tensor operation is the bottleneck operation in tensor factorization (e.g., recommender systems) [4].

We apply Chisel compiler [3] to generate Verilog RTL. For FPGA platform, we use Xilinx Vivado to synthesize the bitstream. For ASIC implementation, we use Synopsys Design Compiler to estimate the area and energy of under the UMC 55nm technology.

B. IR-based Exploration and Tradeoff

As aforementioned, the access entry describes the memory ports and PE interconnection, which further determines the required scratchpad bandwidth. Therefore, Rubick can be used to explore various trade-offs among different hardware implementations by analyzing the access entry of the dataflows.

Fig. 6 presents the trade-off between latency and fan-in/fan-out, where each point is a complete dataflow for GEMM. The axis means different access entry choices, while the color of points represents the latency on the left and the fan-in/fan-out wires on the right. The dataflows in group I require fewer wire resources, but with the longest latency, as most tensors apply type-(a) or type-(b) access entry (refer to Fig. 3). These two types show systolic movements, which need fewer memory ports but take more cycles to load/store input/output data. The dataflows in group III mainly feature with multicast access entry types, which lead to lower latency but higher fan-in/fan-out requirements due to more wires connected with the scratchpad. The dataflows in group II are hybrids of group I and III. Overall, Rubick allows users to make a trade-off between wire resources and latency. For example, group I can reduce 82.4% wire resources compared to group III, with only 2.7% latency increase. Note that this is a unique feature of Rubick compared to prior techniques [10], [12], [15], [17]. The decomposition of dataflow exposed with the low-level details like fan-in/fan-out enable efficient and accurate design space exploration with hardware constraints.

C. Design Space Exploration Comparison

Fig. 7 compares the exploration time of Rubick with TENET [17], which contains the complete space and provides a DSE solution. The time is measured on Intel® Core™ i7-1165G7 @ 2.80GHz with 16GB memory. Overall, Rubick accelerates the design space exploration of dataflows by 1.6×10^3X - 1.1×10^5X. The reason is the decomposition of design space efficiently reduces the total design points, and the pruning techniques avoid the exponential growth of complexity as the number of loops grows. The shape of these tensors is not specified, since the size of the PE array and compute precision are vertical to our design space.

D. FPGA Implementation

Table I compares the FPGA performance of Rubick with AutoSA [23], TensorLib [10] and EMS [12] on 2D-CONV. We select the

TABLE I
FPGA PERFORMANCE COMPARISON

	Device	LUT	DSP	BRAM	MHz	GFLOP/s
AutoSA [23]	U250	56%	77%	30%	272	950
TensorLib [10]	VU9P	73%	75%	73%	245	626
EMS-WS [12]	VU9P	76%	73%	53%	301	731
EMS-OS [12]	VU9P	83%	73%	53%	295	717
Rubick	VU9P	37%	56%	10%	322	826

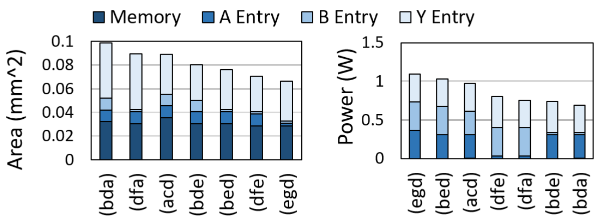


Fig. 8. Area and power breakdown via GEMM dataflow decomposition. The X-axis is different dataflows notated by access entry. E.g., **(bda)** means, tensor A, B, Y applies type-(a), type-(d), type-(b) access entry from Figure 3.

late layers on VGG-19 [21] with FP32 precision as the test bench. We limit the access entry space to suit the features of FPGAs to search for better dataflows. We remove all access entries with a multicast direction vector for the input tensors due to the limited routing resource and improve the frequency by 7%. We select the X-multicast access entry for the output tensor (i.e. adder trees) to avoid data interleaving, which saves BRAM by 5X since only one tile needs to be processed at a time. Rubick also optimized the hardware generation flow. LUT and DSP are further optimized as we can fully analyze the data movement thus simplifying the control logic by avoiding handshaking and additional FIFOs. Overall, we improve the peak performance by 13% compared with EMS-WS [12].

E. Area and Power on ASIC

We also synthesize the RTL for ASIC designs. Fig. 8 (a) and (b) present the area and power breakdown of various GEMM dataflows on an 8×8 PE array with 16-bit integer arithmetic on ASIC. We observe that the output access entry accounts for the most area as it needs to implement reduction operations (e.g., adder tree, accumulators). Dataflows with multicast entries require less area as they only need wires to broadcast data. The memory power is negligible due to the small PE array size. Multicast entries require more energy due to their large fan-out. Stationary entries type-(d) are the most energy-saving one as their registers are idle in most cycles.

VII. CONCLUSION

In this work, we propose Rubick, a synthesis framework for spatial architecture. Our dataflow decomposition features two intermediate representations *access entry* and *data layout*, which formally and systematically provide the implementation details of spatial architecture. We also propose a complete dataflow design space by separately forming the subspace of these two intermediate representations. Based on these IRs we proposed the Rubick synthesis flow for design space exploration and hardware generation, which accelerates the DSE time of dataflows by up to 1.1×10^5X, compared to TENET [17]. Finally, Rubick enables various low-level implementation optimizations for certain hardware platforms, and improves the performance of 2D-CONV on FPGA by 13%, compared to EMS [12].

ACKNOWLEDGEMENTS

This work was supported in part by the National Key R&D Program of China under Grant (No. 2022YFB4500401) and National Natural Science Foundation of China under Grant (No. 61825205).

REFERENCES

[1] M. Abadi, P. Barham, J. Chen *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proceedings of OSDI*, 2016.
[2] A. Anandkumar, R. Ge, D. Hsu *et al.*, “Tensor decompositions for learning latent variable models,” *The Journal of Machine Learning Research*, 2014.
[3] J. Bachrach, H. Vo, B. Richards *et al.*, “Chisel: constructing hardware in a scala embedded language,” in *Proceedings of DAC*, 2012.
[4] J. Bennett and S. Lanning, “The netflix prize,” in *Proceedings of KDD cup and workshop*, 2007.
[5] Y.-H. Chen, T.-J. Yang, J. Emer *et al.*, “Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.
[6] J. Devlin, M.-W. Chang, K. Lee *et al.*, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *NAACL-HLT*, 2019.
[7] H. Genc, S. Kim, A. Amid *et al.*, “Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration,” in *Proceedings of DAC*, 2021.
[8] K. Hegde, P.-A. Tsai, S. Huang *et al.*, “Mind mappings: enabling efficient algorithm-accelerator mapping space search,” in *Proceedings of ASPLOS*, 2021.
[9] Q. Huang, A. Kalaiah, M. Kang *et al.*, “Cosa: Scheduling by constrained optimization for spatial accelerators,” in *Proceedings of ISCA*, 2021.
[10] L. Jia, Z. Luo, L. Lu *et al.*, “Tensorlib: A spatial accelerator generation framework for tensor algebra,” in *Proceedings of DAC*, 2021.
[11] L. Jia, Z. Luo, L. Lu *et al.*, “Automatic generation of spatial accelerator for tensor algebra,” *to appear in the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(TCAD)*, 2022.
[12] L. Jia, Y. Wang, J. Leng *et al.*, “EMS: efficient memory subsystem synthesis for spatial accelerators,” in *Proceedings of DAC*, 2022.
[13] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” *SIAM review*, 2009.
[14] H. Kwon, P. Chatarasi, M. Pellauer *et al.*, “Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach,” in *Proceedings of MICRO*, 2019.
[15] H. Kwon, P. Chatarasi, V. Sarkar *et al.*, “Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings,” *IEEE Micro*, 2020.
[16] Y.-H. Lai, H. Rong, S. Zheng *et al.*, “SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs,” in *Proceedings of ICCAD*, 2020.
[17] L. Lu, N. Guan, Y. Wang *et al.*, “Tenet: A framework for modeling tensor dataflow based on relation-centric notation,” in *Proceedings of ISCA*, 2021.
[18] A. Parashar, P. Raina, Y. S. Shao *et al.*, “Timeloop: A systematic approach to dnn accelerator evaluation,” in *Proceedings of ISPASS*, 2019.
[19] M. Pellauer, Y. S. Shao, J. Clemons *et al.*, “Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration,” in *Proceedings of ASPLOS*, 2019.
[20] A. Radford, J. Wu, R. Child *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, 2019.
[21] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *arXiv preprint arXiv:1409.1556*, 2014.
[22] N. Srivastava, H. Rong, P. Barua *et al.*, “T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations,” in *Proceedings of FCCM*. IEEE, 2019.
[23] J. Wang *et al.*, “Autosa: A polyhedral compiler for high-performance systolic arrays on fpga,” in *Proceedings of FPGA*, 2021.
[24] R. Xu, Y. Xiao, J. Luo *et al.*, “HECTOR: A multi-level intermediate representation for hardware synthesis methodologies,” in *Proceedings of ICCAD*, 2022.
[25] X. Yang, M. Gao, Q. Liu *et al.*, “Interstellar: Using halide’s scheduling language to analyze dnn accelerators,” in *Proceedings of ASPLOS*, 2020.