This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

Rubick: A Unified Infrastructure for Analyzing, Exploring, and Implementing Spatial Architectures via Dataflow Decomposition

Liqiang Lu, Zizhang Luo, Size Zheng, Jieming Yin *Member IEEE*,
Jason Cong *Fellow IEEE*, Yun Liang *Senior Member IEEE*, Jianwei Yin *Member IEEE*

***Abstract*—The fast-growing tensor applications expose tremendous dataflow alternatives when implemented on spatial architectures that feature large PE arrays and abundant interconnection resources. Prior works develop various notations and performance models for dataflows. Though these notations are very useful for understanding the reuse, bandwidth, and performance of dataflows, they do not define the underlying hardware implementation. Due to the semantic gap, analysis based on these notations cannot capture the detailed architectural features between different dataflows, leading to inefficient design space exploration and suboptimal designs.**

To address these issues, we propose Rubick, a unified infrastructure for analyzing, exploring, and implementing spatial architectures. The main innovation of Rubick is it decomposes the dataflow into two low-level intermediate representations: *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. These two representations allow us to infer the hardware implementation details such as PE interconnection and memory structure, which are amenable for structural analysis and systematic exploration. Based on this decomposition analysis, Rubick provides opportunities for micro-architecture optimization and efficient design space exploration. Our experiments demonstrate that Rubick can reduce 82.4% of wire resources with only a 2.7% latency increase by optimizing access entry IR, and achieve 70.8% memory overhead reduction by optimizing data layout IR. Rubick also accelerates the DSE time of dataflows by up to 1.1×10^5X, saving the time from several days to minutes. The source code of Rubick is publically available on (https://link-omitted-for-blind-review).

I. Introduction

Spatial architectures play a pivotal role in the acceleration of various tensor applications [6]–[9], [13], [16], [17], [21], [23], [31], [37], [45], [51], [53]–[56], [64], [66], [73]. 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 [1], [2], [14], [27], [42], [59], [62], [67].

Hardware dataflow is the key component when implementing applications onto spatial architectures, which assigns the instance in the *loop iteration domain* to a spacetime-stamp in the *dataflow spacetime domain*. Specifically, the space-stamp gives the PE location to execute an instance, while the time-stamp determines the execution sequence. Therefore, the dataflow implies 1) how data enter the specific PEs from the scratchpad SRAM and traverse across PE array, 2) how data are arranged in the on-chip memory and scheduled during computation. For example, Google's Tensor Processing Unit (TPU) applies systolic dataflow to accelerate general-purpose matrix multiplication (GEMM). This dataflow determines that only boundary PEs will read data, and data traverse between adjacent PEs. Besides, the data layouts are skewed when accessed from the scratchpad to PEs. While Cambricon [38] and MAERI [31] feature reduction tree dataflow, which indicates that data are broadcast to PEs in rectangle data layout. Other spatial architectures that enable reconfigurability like DySER [17] and Plasticine [54], integrate PEs and their interconnection in a flexible manner and hence support a wider range of dataflows.

Recently, several frameworks have been proposed for dataflow analysis and performance modeling [5], [8], [18], [20], [25], [29], [30], [39], [40], [43], [49], [50], [70]–[72]. Among them, MAESTRO [29], Interstellar [71], and TENET [39] are three state-of-the-art dataflow modeling frameworks. MAESTRO [29] proposes a data-centric notation that represents the dataflow according to the data index allocation. Interstellar [71] uses loop-nest with primitives to describe the dataflow, known as the compute-centric notation. TENET [39] proposes a relation-centric notation that models dataflows as mappings from computational instances to PEs and cycles. Despite that all these frameworks are capable of precisely modeling the dataflow and estimating performance metrics like reuse and latency, there is still a semantic gap between these dataflow notations and architecture implementation. 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

Liqiang Lu and Jianwei Yin are with College of Computer Science, Zhejiang University, Hangzhou, Zhejiang, 310058 China. Zizhang Luo, Size Zheng, Yun Liang are with Center for Energy-efficient Computing and Applications, Peking University, Beijing, 100871 China. Jieming Yin is with School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu, 210003 China. Jason Cong is with Computer Science Department, University of California at Los Angeles, CA, 90095 U.S.. Jianwei Yin and Yun Liang are the corresponding authors. Liqiang Lu and Zizhang Luo contributed equally.
E-mail: {liqianglu,zjuyjw}@zju.edu.cn,
{semiwaker,zhengsz,ericlyun}@pku.edu.cn,
jieming.yin@njupt.edu.cn, cong@cs.ucla.edu.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

of each tensor from the high-level notation, including data access patterns and data arrangements. On the other hand, the spatial architecture consists of PEs and memory, which serve distinct purposes in program execution. The architecture implementation of computation and memory requires different low-level architectural features.

Another limitation of prior frameworks is inefficient design space exploration. The design space formed by alternating the parameters at high level is explosively large, which takes extremely long to explore. Furthermore, we observe that some dataflows generated by prior frameworks are inferior due to low resource utilization and redundant computation. In addition, the structural similarity between different dataflows cannot be recognized at high level. For example, TPU [23] and OuterSpace [48] are two distinguished GEMM dataflows that exhibit different parallelism. However, they share the same data movement of one input matrix, which actually can be inferred from low-level representations. As a result, the design space exploration at high level cannot be used for hardware optimization when architectural constraints are specified.

In this paper, we propose Rubick, a unified infrastructure for analyzing, exploring, and implementing spatial dataflow. **To analyze the dataflow**, we first decompose the dataflow as a Cartesian product of different tensor movements. Then, each tensor movement is further decomposed into a chain product of two low-level IRs: *access entry* and *data layout* to describe the hardware characteristics of computation and memory, respectively, which can be beneficial to abstract some hardware details. To be more concrete, the access entry explicitly provides PE interconnection and memory interface, describing both the location and timing of data transfers from memory to PE. The data layout describes which element is used for a specific access entry and thus explicitly specifies the tensor data arrangement in the memory bank of the on-chip buffers, and its access sequence as address generators. With these two low-level IRs, Rubick can expose rich architectural details.

To explore the dataflow, we form the design space of dataflows in a structured way. We first form the sub-space of access entry and data layout separately, then compose them together. By doing this, we can easily capture the similarity among the dataflows within each subspace, thus dramatically reducing the total space by pruning out hardware inefficient designs. More clearly, the access entry space is formed as a linear space that consists of multiple linear combinations of access direction vectors. The data layout space enumerates all the possible linear transformation that maps the tensor to a spacetime-stamp.

To implement the dataflow, we present the relationship between Rubick IRs and the implementation details. To be specific, we demonstrate how the access entry IR determines the PE micro-architecture, e.g., fan-in/fan-out, pipeline latency, and reduction scheme. The memory hierarchy is implemented as multi-dimensional times-stamps of data layout IR, including off-chip memory, on-chip memory and address generator. Finally, we develop a generation tool that can automatically implement the hardware using IR Chisel template.

A preliminary version of this paper will appear in DAC 2023 [41], we proposed to synthesize various dataflow through dataflow decomposition. In this article, we extend previous work with analytical techniques to further demonstrate the benefit of Rubick IRs. To be specific, we present the intuition of why there requires a decomposition theory, and provide further architecture implementations including hardware optimizations and hardware generation. Finally, we apply our decomposition technique on various dataflow to extract the low-level information and provide optimization results after balancing the different trade-off.

In conclusion, this work makes the following contributions,

- We propose dataflow decomposition into IRs for analyzing the dataflow, which are formulated as integer mapping functions that explicitly expose low-level architecture.
- We propose a systematic and efficient dataflow formulation methodology that composes the dataflow in the sub-space of each IR, which supports to search dataflow under low-level constraints.
- We propose the methodology for dataflow implementation using Rubick IRs. By closing the semantic gap between dataflow and architecture, Rubick allows various optimization techniques and hardware generation.

Our experiments demonstrate that Rubick can reduce 82.4% wire resources with only 2.7% latency increase by optimizing access entry IR. For multi-kernel benchmark, Rubick shows 5.6X - 49X, 64X reduction for intermediate buffer size compared to NVDLA [47] and TPU [23]. 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 [39].

II. Background

A. Tensor Basics

A tensor is defined as matrices with 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 indexes.

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

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.

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

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

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 Dataflow

A key component of a spatial architecture is the dataflow that determines how a tensor kernel is mapped onto the architecture. In general, the dataflow is represented from two aspects: 1) the space-stamp that describes where a loop instance is executed, and 2) the time-stamp that describes when a loop instance is executed. In this paper, we assume that the space-stamp refers to the PE, and the time-stamp refers to the execution cycle. Various notations have been proposed recently, including compute-centric notation [71], data-centric notation [29], [30], and relation-centric notation [39]. In this paper, we choose to use the relation-centric notation, as it is more expressive than the other two notations and can express the complete design space of dataflows. Using relation-centric notation, the dataflow is a set of relations where each relation 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 iteration domain to a spacetime-stamp from dataflow spacetime domain. The space-stamp $PE(\vec{p})$ gives the coordinates of PE where the instance will be executed, and the time-stamp gives the execution sequence. $\vec{t}$ can be one or multi-dimensional and the sequence is determined by the lexicographical order of time-stamp $T(\vec{t})$. For example, $S(0,1,0) \to (1,0 \mid 0,1)$ means the instance $S(0,1,0)$ is executed in PE(1,0) at cycle(0,1).

Tensor movement. 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. For a specific dataflow spacetime-stamp, it gives the required tensor element.

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

C. Motivation

Though prior dataflow frameworks [29], [30], [39], [71] can accurately estimate performance metrics such as data reuses, latency, etc., the semantic gap between these high-level notations and low-level hardware renders it impossible to infer the architectural implementation details and perform structural analysis based on these notations. Figure 1 gives three 1D-CONV dataflow examples with their low-level architecture implementations. The 1D-CONV instance is written as follows.

$$S(i,j): \quad Y(i) += A(i + stride \cdot j) \times B(j) \quad (4)$$

To make a difference, we set the stride of dataflow (a), (b), (c) as 1, 2, 1, respectively. In each dataflow, four instances in the yellow parallelogram are executed simultaneously at the first cycle($t=0$) on a 2×2 PE array. The green parallelogram is executed at the second cycle($t=1$).

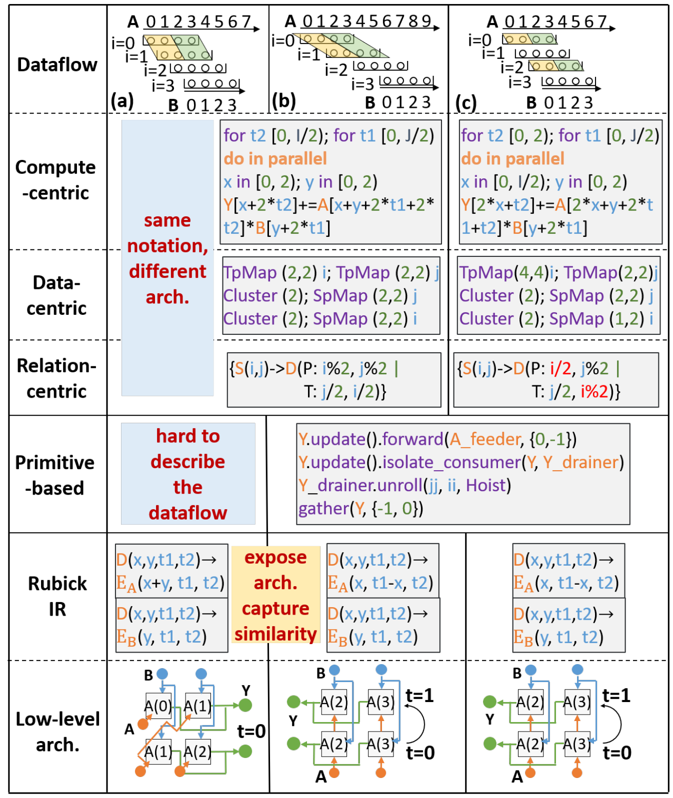


Fig. 1. Motivational example using 1D-CONV. Prior compute-centric [20], [71], data-centric [24], [29], relation-centric [21], [39] approaches cannot expose the low-level architecture, while primitive-based approach cannot explicitly describe the dataflow. Rubick IR can bridge the semantic gap between dataflow and architecture.

First, prior notations do not directly describe the hardware details. We observe that dataflow (a) and (b) share the same dataflow notation, however, have different architectures. To be specific, the data-centric notation [29], [30] represents the dataflow by spatially allocating 2 elements of tensor Y (`SpMap(2,2) i`), and 2 elements of tensor B (`SpMap(2,2) j`). Using the relation-centric notation, two continuous index i is horizontally mapped to the PE array, and two continuous index j is vertically mapped. Thus, the dataflow is written as $\{S(i,j) \to (PE(i\%2, j\%2) \mid T(i/2, j/2))\}$. To implement the dataflow, in dataflow (a), the architecture shows a diagonal datapath of tensor A. While dataflow (b) requires a vertical datapath. Such low-level architectural features cannot be exposed by the notations. Our access entry IR supports this, e.g., $(x+y)$ means diagonal access direction, $(t1 - x)$ means vertical streaming in different cycles.

On the other hand, the notation of dataflow (b) and (c) is different. For example, using relation-centric notation, two indexes of i with an interval are horizontally mapped to the PE array. the dataflow is written as $\{S(i,j) \to (PE(i/2, j\%2) \mid T(i\%2, j/2))\}$. However, the architecture of (b) is the same as (c), which means these notations cannot capture the similarity between different dataflows, leading to inefficiency of design space exploration. While in our

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

approach, we can clearly see the IRs of these two dataflows are the same, resulting in the same implementation.

There are primitive-based representations for describing the hardware implementation, like HeteroCL [32], Susy [33], T2S [63], AutoSA [68]. These approaches are essentially generation tools that adopt high-level languages and rely on high level synthesis (HLS) to generate hardware. Figure 1 shows the T2S representation for dataflow (b) and (c). Though it tells the implementation using hardware primitives, this approach fails to model the dataflow behavior, e.g., when and where a tensor element is used. Besides, they only cover a subset of dataflow space. For example, it is hard to write the primitive for non-orthogonal dataflow like (a).

The semantic gap between dataflow and architecture fundamentally results from the fact that 1) the computational instance is a mixture of multiple tensor behavior, which needs to be explicitly represented for architecture implementation, 2) the spatial architecture involves PE array part and memory, which serve distinct purposes for tensor application execution. Based on these two insights, we propose dataflow decomposition that decomposes the dataflow into tensor movements, and then decomposes the tensor movements into low-level architectural IRs.

III. Dataflow Analysis via 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 efficient design space exploration. By defining rigorous architectural constraints for the subspace corresponding to each IR, the combined space can be significantly pruned. We derive these two IRs by defining a new domain called the *entry spacetime domain*, as shown in Figure 2.

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.

- The left domains (iteration and tensor) in Figure 2 are tensor application-driven domains, which are constituted with loop instances and tensor data. The right domains (dataflow and entry) are architecture-driven domains, which are constituted with space-time stamps. Dataflow (Θ) maps loop instances to space-time stamps, while data layout (L) maps space-time stamps to tensor elements.
- The top domains (iteration and dataflow) are loop instance-driven, while the bottom domains (tensor and entry) are tensor data-driven. The access function (A) bridges the tensor data with loop instances. On the other hand, the dataflow spacetime domain represents a complete architecture, including PE units, PE interconnection, and memory. The access entry (Ω) decouples these implementation details from spatial architecture.
- Entry spacetime domain is both architecture-driven and data-driven domain. It is the interface between PE array and memory, which controls where and when one tensor element is accessed. Thus, *access entry* provides the datapath. *data layout* provides tensor data arrangement and access sequence.

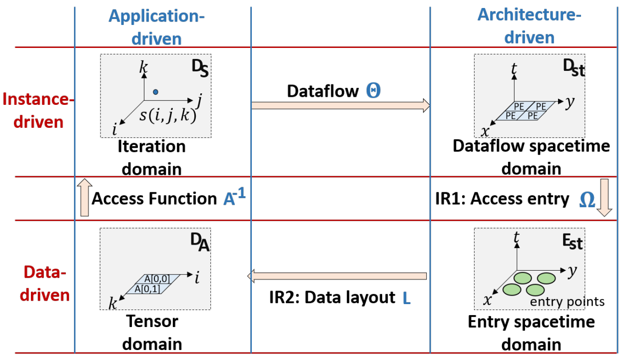


Fig. 2. Domains in dataflow decomposition.

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 for architecture implementation (Section III-B).

A. Dataflow Decomposition

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

$$\Theta_{D_{st}\to D_S} = (M_{D_{st}\to D_A} \otimes M_{D_{st}\to D_B}, \ldots) \times \mathcal{A}_{(D_A, D_B, \cdots)\to D_S} \tag{5}$$

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,

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

As shown in Figure 2, 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}.

$$\Omega_{D_{st}\to E_{st}} = \{(PE(\vec{p_d}) \mid T(\vec{t_d})) \to (E(\vec{p_e}) \mid T(\vec{t_e}))\} \tag{6}$$

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.

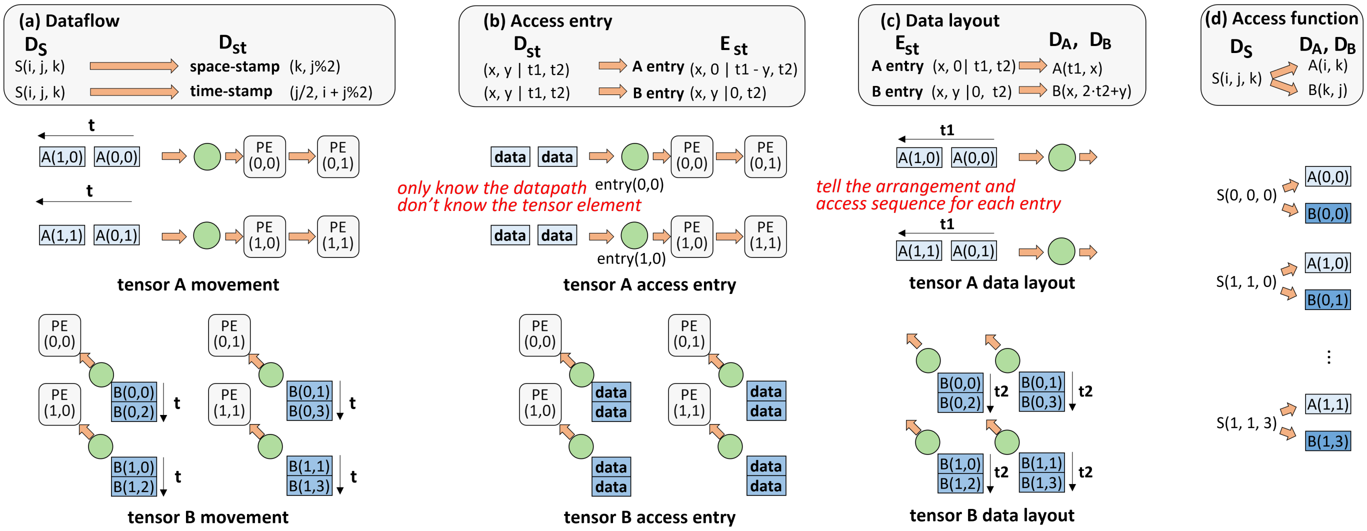


Fig. 3. GEMM dataflow decomposition example. The dataflow is decomposed into access entry IR and data layout IR.

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,

$$L_{E_{st}\to D_A} = \{(E(\vec{p_e}) \mid T(\vec{t_e})) \to A(\vec{n'})\} \tag{7}$$

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 access sequence of the tensor to or 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.

$$M_{D_{st}\to D_A} = \Omega^A_{D_{st}\to E_{st}} \times L_{E_{st}\to D_A} \tag{8}$$

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

$$\begin{aligned}\Theta_{D_{st}\to D_S} = &\left(\Omega^A_{D_{st}\to E_{st}} \times L_{E_{st}\to D_A}\right)\\ &\otimes \left(\Omega^B_{D_{st}\to E_{st}} \times L_{E_{st}\to D_B}\right)\\ &\times \mathcal{A}_{(D_A,D_B)\to D_S}\end{aligned} \tag{9}$$

B. Decomposition Example

In this subsection, we use GEMM dataflow as an example to illustrate dataflow decomposition. As shown in Figure 3 (a), the dataflow is written as follows.

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

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). For simplicity, we write the dataflow spacetime-stamp D_{st} in Figure 3 (a) as $\{(x, y \mid t1, t2)\}$.

Then, we formulate the access entry of input tensor A and tensor B, as shown in Figure 3 (b).

$$\textbf{Tensor A}\quad \Omega^{\mathrm{A}}_{\mathrm{D_{st}\to E_{st}}} = \{(x, y \mid t1, t2) \to (x, 0 \mid t1 - y, t2)\}$$
$$\textbf{Tensor B}\quad \Omega^{\mathrm{B}}_{\mathrm{D_{st}\to E_{st}}} = \{(x, y \mid t1, t2) \to (x, y \mid 0, t2)\}$$

Identifying that the entry space-stamp is a 1D-vector (the second dimension of 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 access entry of tensor B shows the same spatial distribution as the PE array but different time-stamps. The first dimension of the entry time-stamp is 0, resulting in reduced memory requirements as tensor B remains static in the PE register until the second time dimension $t2$ changes.

In Figure 3 (c), we provide the data layout of both tensors A and B with the spatial distribution of entries.

$$L_{E_{st}\to D_A} = \{(E(x, 0) \mid T(t1, t2)) \to A(t1, x)\}$$
$$L_{E_{st}\to D_B} = \{(E(x, y) \mid T(0, t2)) \to B(x, 2 \cdot t2 + y)\}$$

Note that, the access entry IR only tells there is a data accessed 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). Moreover, by analyzing which dimension of tensor A is mapped to the space-stamp or the time-stamp, we can know the arrangement of tensor elements in the memory, and its access sequence.

IV. Dataflow Design Space Exploration

For a given dataflow, we can specify one of them and calculate another according to Equation 8. 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

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

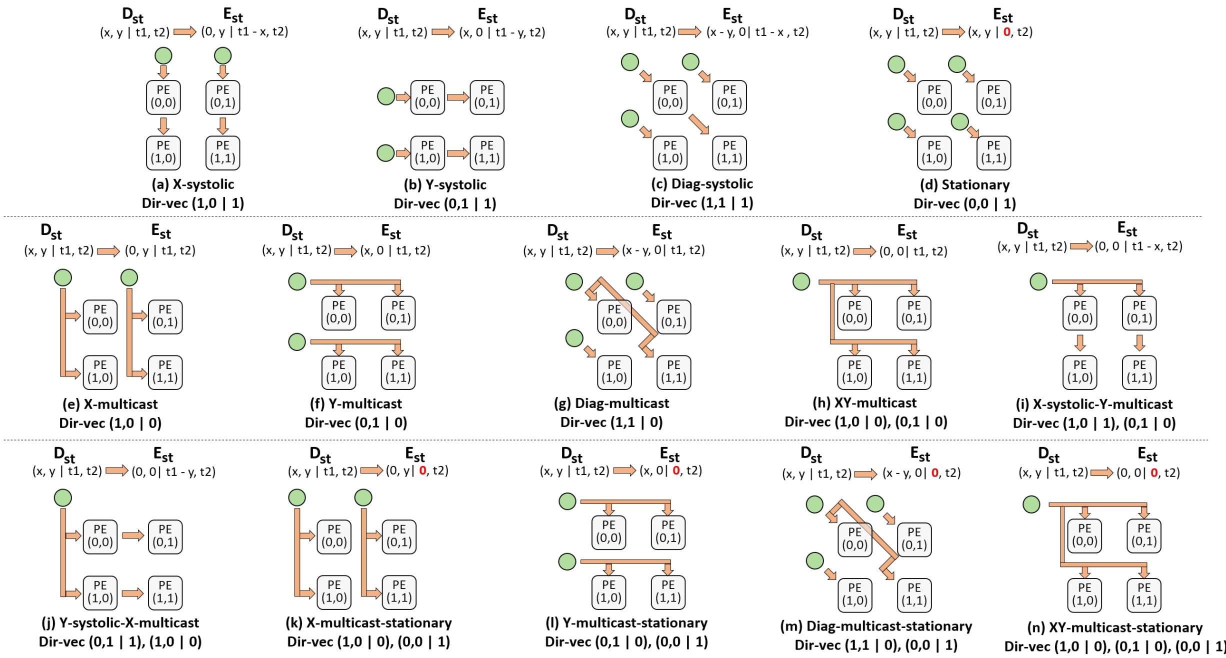


Fig. 4. Input access entry space on 2D-PE array. The space is formulated as tensor access direction vectors.

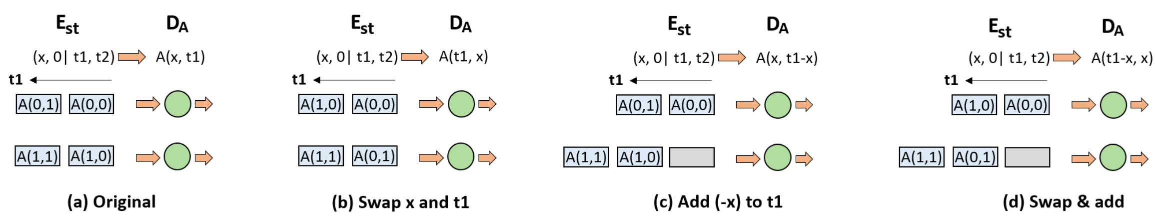


Fig. 5. Data layout space on 2D-PE array. The space is formulated as linear matrix transformation.

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 (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 former 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. Figure 4 lists all of them on a 2D-PE array. Figure 4 (a)-(c) are systolic patterns with horizontal, vertical and diagonal (slope $= 1$) data transfer. In Figure 4 (d), the first dimension of time-stamp is 0, representing each PE keeps the tensor element stationary for a while. Figure 4 (e)-(g) are multicast networks where entries spatially distribute like a 1D-vector. The last six access entries in Figure 4 (i)-(n) are hybrid patterns.

Note that Figure 4 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

Our target architecture reads data from on-chip SRAM buffers into the PE array, with the data layout space being dependent on both the application (tensor domain) and the 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. E.g., in Figure 3 (c), the data layout of tensor B has a coefficiency of 2. Due to the smaller size of the PE array (2×2) compared to the size of tensor B (2×4), the second dimension of tensor B needs to be tiled (i.e. is cut into smaller blocks, resulting in a size of $2 \times 2 \times 2$). The tensor access behavior mainly depends on the first two transformations. Figure 5 depicts how

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

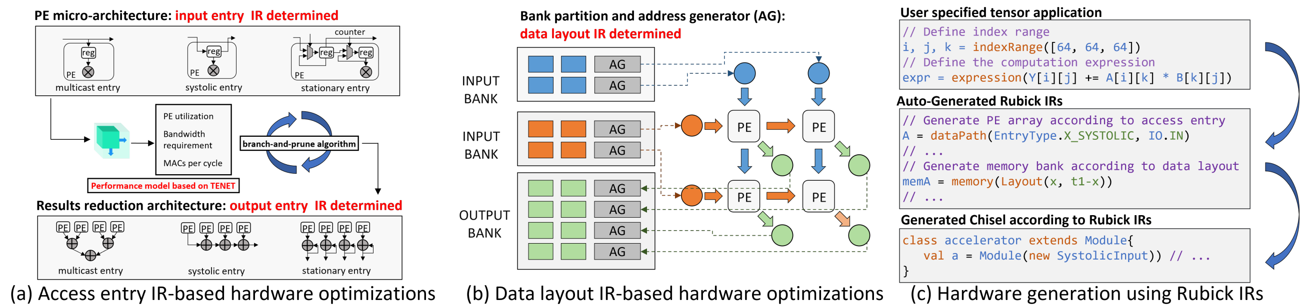


Fig. 6. Architecture implementation using Rubick IRs.

the linear transformation affects the data layout. Figure 5 (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 Figure 5 (a), it acts like a transposition when tensor A is a matrix. In Figure 5 (c), we add $(-x)$ to $t1$ in E_{st} and map it to D_A, leading to data skewing.

C. Entire Space Formation and Space Pruning

Both spaces are linear transformation spaces formed via linear algebra. The difference lies in that the space of access entry is formulated by direction vectors, which essentially are the basis vectors in the complementary linear space. To find out the correct access entry during decomposition, we only need to test direction vectors and select the vectors that satisfy $M_{D_{st}\to D_A}(\vec{r}) = 0$. For example, we only need to test the 7 direction vectors for a 2D-PE array, and it is 15 for a 3D-PE array. Besides, the architectural constraint, like interconnection topology, will affect the choice of these vectors during dataflow exploration. On the other hand, the space of data layout is formulated by a linear transformation. The space is determined by both PE dimensions and tensor dimensions. For example, assuming that we only apply linear transformation in two dimensions, then, the data layout number of a 4D-tensor is $C_4^2 \times 4 = 24$, where 4 means the four types in Figure 5.

By separately constructing the sub-space of each IR, the total design space is dramatically reduced. We establish the performance model for memories and bandwidths using methods similar to TENET [39]. We observe that employing a simple branch-and-prune algorithm is sufficient to search the entire design space within a reasonable time. Besides, we also propose three general pruning strategies to further reduce the overall space. The first one prunes the point that involves non-full-rank mapping from the entry spacetime domain to the dataflow spacetime domain. A non-full-rank mapping leads to multiple data mapping to one entry point at one cycle. After selecting the IRs of input tensors, we can obtain the movement of output tensor. The second pruning strategy prunes the points with wrong output results. The wrong output is due to the unmatched tensor movement, specifically, unmatched access entry or data layout. Similarly, the last pruning strategy will check whether the final dataflow meets the full-rank constraint.

V. Dataflow Implementation

A. PE Architecture Implementation

As mentioned in Section III-A, access entry IR describes the spatial location of entry points for data transfer between tensor and PE, where each entry point corresponds to one memory bank. The time information in access entry IR implies the data transfer direction, which further determines the PE interconnection topology. Figure 6 (a) shows the PE micro-architecture of different types of access entry. Multicast entries require broadcast wires between memory and PEs, without inter-PE connections. These entries feature large fan-in or fan-out but low pipeline latency. On the other hand, systolic entries have the minimum fan-in or fan-out but have longer latency to deliver data to all PEs. Using the stationary entry, each PE loads data individually from different addresses, thus exhibiting no interconnection. The architecture of the reduction module is determined by the output access entry IR. For example, the systolic entry only loads one result at each cycle, and accumulates them via an adder array. The output stationary entry updates iteratively in a local register.

B. Memory Implementation

As shown in Figure 6 (b), data layout IR directly determines the memory hierarchy and tensor data layout. Clearly, this IR is responsible for partitioning tensors to different banks and generating the address of each bank. The memory hierarchy is modeled as multi-dimensional timestamps. In this model, the innermost time dimensions intricately depict the behavior of on-chip memory, while the outer dimensions aptly capture the memory behavior exhibited by DRAM or host memory. For example, we can label the time-stamp as $(T(\textit{PE register file}), T(\textit{on-chip SRAM}), T(\textit{off-chip DRAM}))$. Based on the tensor index range, we can get the range of each time-stamp, which further determines the memory size. In our experiments, we provide the optimization of intermediate buffer size for multi-kernel applications.

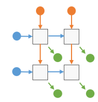
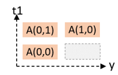
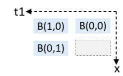
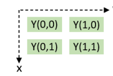
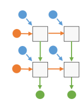
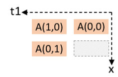
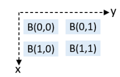
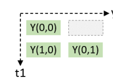
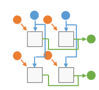
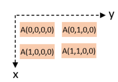
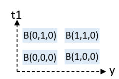
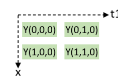
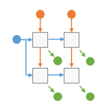
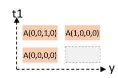
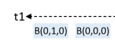
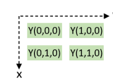
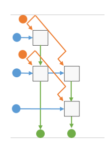
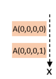
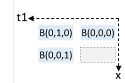
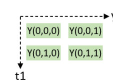
C. Hardware Generation

As Rubick IRs explicitly expose implementation details, we develop an automatic hardware generation tool using Chisel [3] templates, as shown in Figure 6 (c). The generator takes tensor computation expressions specified by index range as inputs and generates a complete hardware design. First, it decomposes the dataflow into IRs. The dataflow can be specified by the user or searched from the design space. Then, we can generate the datapath logic based on access entry IRs, which have two sets of different templates, depending on whether the tensor is input or output. Note that IRs can also be specified by users. Finally, the data layout IRs help

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

TABLE I

DATAFLOW DECOMPOSITION VISUALIZATION. ALL DATAFLOWS ARE MODELED ON A 8×8 PE ARRAY. IN THE ACCESS ENTRY, A-(A) MEANS TENSOR A IS ACCESS IN TYPE-(A) PATTERN OF FIGURE 4. IN THE DATA LAYOUT, WE ONLY PICTURE FOUR TENSOR ELEMENTS FOR SIMPLICITY. DIFFERENT COLOR REPRESENTS DIFFERENT TENSOR. YELLOW: TENSOR A. BLUE: TENSOR B. GREEN: TENSOR Y.

Benchmark	Dataflow	Access Entry	Data Layout		
GEMM	**GEMM-(a)** S(i,j,k)→PE(j%8,i%8) S(i,j,k)→T(i%8+j%8+k,⌊i/8⌋,⌊j/8⌋) motivated by OuterSpace [48]	A-(a),B-(b),Y-(d)	E(0,y ∣ t1,t2,t3)→ A(y+8·t2,-y+t1)	E(x,0 ∣ t1,t2,t3)→ B(-x+t1,x+8·t3)	E(x,y ∣ 0,t2,t3)→ Y(y+8·t2,x+8·t3)
					
	GEMM-(b) S(i,j,k)→PE(k%8,j%8) S(i,j,k)→T(i+j%8+k%8,⌊j/8⌋,⌊k/8⌋) applied in TPU [23]	A-(b),B-(d),Y-(a)	E(x,0 ∣ t1,t2,t3)→ A(-x+t1,x+8·t3)	E(x,y ∣ 0,t2,t3)→ B(x+8·t3,y+8·t2)	E(0,y ∣ t1,t2,t3)→ Y(-y+t1,y+8·t2)
					
2DCONV	**2DCONV-(a)** S(k,c,ox,oy,rx,ry)→PE(k%8,c%8) S(k,c,ox,oy,rx,ry)→T(ox,oy,rx,ry,⌊k/8⌋,⌊c/8⌋) applied in DianNao [6] and NVDLA [47]	A-(d),B-(e),Y-(f)	E(x,y ∣ 0,t2,t3,t4,t5,t6)→ A(x+8·t5,y+8·t6,t3,t4)	E(0,y ∣ t1,t2,t3,t4,t5,t6)→ B(y+8·t6,t1+t3,t2+t4)	E(x,0 ∣ t1,t2,t3,t4,t5,t6)→ Y(x+8·t5,t1,t2)
					
	2DCONV-(b) S(k,c,ox,oy,rx,ry)→PE(ox%8,k%8) S(k,c,ox,oy,rx,ry)→ T(k%8+ox%8+rx,c,oy,ry,⌊k/8⌋,⌊ox/8⌋)	A-(a),B-(j),Y-(d)	E(0,y ∣ t1,t2,t3,t4,t5,t6)→ A(y+8·t5,t2,-y+t1,t4)	E(0,0 ∣ t1,t2,t3,t4,t5,t6)→ B(t2,t1+8·t6,t3+t4)	E(x,y ∣ 0,t2,t3,t4,t5,t6)→ Y(y+8·t5,x+8·t6,t3)
					
	2DCONV-(c) S(k,c,ox,oy,rx,ry)→PE(oy%8+ry%8,oy%8) S(k,c,ox,oy,rx,ry)→ T(ox+oy%8+ry%8,k,c,rx,⌊oy/8⌋,⌊ry/8⌋)	A-(m),B-(f),Y-(a)	E(x,0 ∣ 0,t2,t3,t4,t5,t6)→ A(t2,t3,t4,x+8·t6)	E(x,0 ∣ t1,t2,t3,t4,t5,t6)→ B(t3,-x+t1+t4,x+8·t5+8·t6)	E(0,y ∣ t1,t2,t3,t4,t5,t6)→ Y(t2,t1,y+8·t5)
					

to generate the memory modules and address generators for data control and data transfer. The index range determines the range of time-stamps, which further determines the memory size. Furthermore, our hardware generator is modular and can be extended for different spatial architecture designs.

VI. EXPERIMENT

This section evaluates Rubick. In Section VI-A, we present the experimental settings, including benchmarks and implementation. Section VI-B presents the analysis results of various dataflow using our decomposition methodology. Section VI-C and Section VI-D provide the exploration results of access entry IR and data layout IR, respectively, including the evaluation of trade-offs between latency, fan-in/fan-out, and memory size. In Section VI-E, we compare the DSE exploration efficiency with the state-of-the-art modeling framework TENET [39]. Finally, we show the implementation results on ASIC (Section VI-F and VI-G) to demonstrate that Rubick can perform various architectural optimizations.

A. Experiment Setup

Benchmarks. We evaluate the following benchmarks.

$$\begin{array}{ll} \textbf{GEMM} & Y(i,j) += A(i,k)B(k,j) \\ \textbf{2D-CONV} & Y(n,k,ox,oy) += A(k,c,rx,ry)B(n,c,ox+rx,oy+ry) \\ \textbf{MMc} & Y(i,j) += A(i,k)B(k,l)C(l,j) \\ \textbf{MTTKRP} & Y(i,j) += A(i,k,l)B(k,j)C(l,j) \end{array} \tag{10}$$

GEMM and 2D-CONV are single kernels, which are widely used in deep learning and scientific computing [1], [2], [27], [62]. Matrix multiplication chain (MMc) is used in the attention mechanism of transformer models [12], [34], [57]. Matricized tensor times Khatri-Rao product (MTTKRP) tensor operation is the bottleneck operation in tensor factorization (e.g., recommender systems) [4], [44].

implementation. Rubick support two implementation backends, including ASIC and FPGA platform. We use Chisel cycle-accurate simulator to evaluate the performance. We apply Chisel compiler [3] to generate Verilog RTL. For ASIC implementation, we use Synopsys Design Compiler to estimate the area and energy of under the UMC 55nm technology. For FPGA platform, we then use Xilinx Vivado to synthesize the bitstream for FPGA implementation. Empirically, we assign the first two time dimensions of data layout IR to on-chip

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

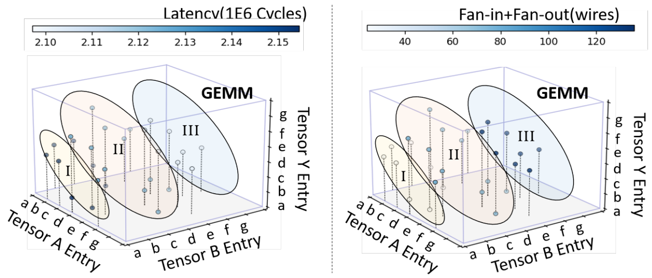


Fig. 7. Using Rubick access entry IRs to explore hardware design. a-g means the access entry type in Figure 4.

BRAMs, with the rest scheduled to off-chip DRAM.

B. Dataflow Analysis via Decomposition

For the GEMM benchmark, we use Rubick to analyze two popular dataflows that applied in TPU [23] and OuterSpace [48]. We visualize the dataflow decomposition of each tensor, and analyze the architecture implementation, as shown in Table I. According to the access entry of GEMM-(a) dataflow, we can clearly know that both two input tensors are stored in 1D banks, and each PE is responsible for a different output element. We can also understand how to schedule the data according to the data layout. Clearly, two input tensors are accessed with skewing from memory to PEs, while each output element is kept in the PE register until the second time dimension $t2$ changes. The access entry of TPU dataflow indicates that the architecture requires downward accumulators to gather the results and store them to memory via the bottom PEs. The data layout tells that tensor A and output are skewed when scheduling, while tensor B is kept stationary in the PE.

2D-CONV is a much more complex tensor benchmark that involves six loops. We present the decomposition of one common 2D-CONV dataflow and two dataflows found by Rubick, which minimizes the number of memory ports. 2D-CONV (a) dataflow is applied in DianNao [6] and NVDLA [47], leveraging the parallelism in the k and c dimensions that show less data dependency. This dataflow requires 8 memory ports in total where tensor B is vertically broadcast to PEs. Different from GEMM-(b), the output access entry of 2D-CONV (a) indicates that multiplication results are generated simultaneously, which means there needs an adder tree to gather the results. To minimize the port number of tensor B, 2D-CONV (b) dataflow specifies tensor B access entry as a scalar (type (j): Y-systolic-X-multicast in Figure 4). Consequently, the data layout of tensor B is expanded only in time dimensions (x==0, y==0). While 2D-CONV (c) dataflow tries to adopt systolic entry or multicast entry for all tensors to minimize the port number. To this end, the PE array is transformed to parallelogram shape where tensor A is diagonally broadcast to PEs and kept stationary, and results are downward accumulated. We also observe that the data layout of tensor B is skewed to match the parallelogram PE array.

C. Access Entry IR-based Exploration

As mentioned in Section III-A, the access entry describes the memory ports and PE interconnection, which further determines the required scratchpad bandwidth. Therefore, Rubick

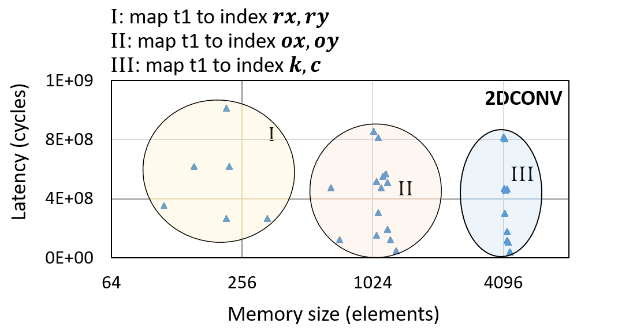


Fig. 8. Using Rubick data layout IRs to explore hardware design. rx, ry, ox, oy, k, c are tensor indices in Equation 10

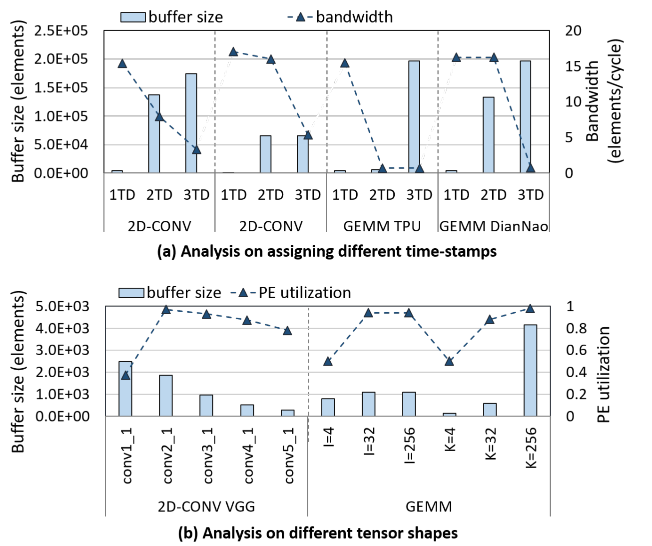


Fig. 9. Analyzing the trade-off between buffer size, bandwidth, PE utilization using data layout IR.

can be used to explore various trade-offs among different hardware implementations by analyzing the access entry of the dataflows, e.g., the trade-off between latency and fan-in/fan-out, latency and memory size, fan-in/fan-out and bandwidth requirement.

Figure 7 illustrates the trade-off between latency and fan-in/fan-out, where each data point represents a complete dataflow of GEMM with a shape of $64 \times 64 \times 64$. The axis means different access entry choices, while the color of points represents the latency on the left of Figure 7 (the darker the longer latency), and represents the fan-in/fan-out wires in the right of Figure 7 (the darker the more fan-in/fan-out), respectively. The dataflows in group I require fewer wire resources, but show the longest latency, as most tensors apply type-(a) or type-(b) access entry (refer to Figure 4). 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 multicast access entry types, which leads 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.

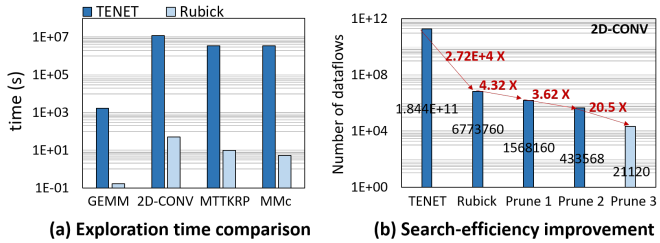


Fig. 10. Exploration-efficiency improved by Rubick.

D. Data Layout IR-based Exploration

Figure 8 illustrates the trade-off between latency and memory size by analyzing the data layout in 2DConv. The input has a shape of $256 \times 64 \times 64$, and the kernel has a shape of $256 \times 256 \times 8 \times 8$. Here we refer memory as the on-chip scratchpad that stores a tile of data for inner-most time-stamp $t1$. Each group here may involve multiple dataflows depending on the linear transformation between $t1$ and tensor indices. In group I, the data layout maps the time dimension to smaller tensors, which has longer latency but requires less memory (e.g., map rx, ry to $t1$). Inversely, the dataflows in group III need more memory but show lower latency. The lowest latency point in group II can reduce 67.8% memory compared to the lowest latency point in group III.

Previously, we assume that only the first time dimension (TD) of data layout IR is assigned to the on-chip memory. Figure 9 (a) shows the trade-off between buffer size and bandwidth when assigning multiple time dimensions. As a result, more on-chip time dimensions lead to a larger buffer size. However, it does not necessarily reduce the bandwidth requirement, depending on whether the dimension provides the data reuse opportunities. For example, 2TD and 3TD cases of TPU have the same bandwidth requirement. This can be explained by the fact that the third time dimension is mapped to the k tensor dimension (GEMM-(b) in Table I), which is a reduction dimension that contributes no reuse.

The tensor index range determines the time range in the data layout IR, which affects the required buffer size and PE utilization. We evaluate different shapes using NVDLA [47] 2DCONV dataflow (2DCONV-(a) in Table I) on VGG network [61], as shown in Figure 9 (b). As shown in the NVDLA dataflow in Table I, the range of inner time dimensions (ox, oy) reduces when the network goes deeper, which causes the required buffer size to decrease. The utilization of CONV1_1 layer is low due to the small input channel size ($c = 3$), resulting in low PE utilization. For GEMM case, OuterSpace [48] dataflow (GEMM-(a) in Table I) adopts outer-product parallelism. Therefore, the $I = 4$ case cannot fully utilizes the PE array. The $K = 4$ case has less reuse opportunities, thus causing high transfer cost and PE array under utilized.

E. Exploration Comparison with TENET

Figure 10 (b) presents the breakdown exploration efficiency for 2DCONV. Since the loop boundary is usually larger than the PE array size, the original six loops are tiled into eight loops with two mapped to the PE array. The space of TENET is huge with many inferior dataflows. We reduce this space

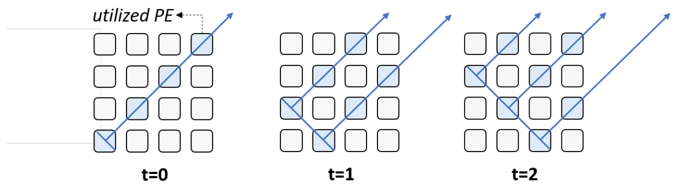


Fig. 11. TENET inferior dataflow with under-utilized PE.

TABLE II
DATA LAYOUT IR OF DIFFERENT DESIGN.

Benchmark	Dataflow	Output Layout	Input Layout
SCONV+ PCONV [19]	NVDLA [47]	E(x,0 | t1,· · · ,t6)→ P(x+8·t6,t1,t2)	E(0,y | t1,· · · ,t4)→ P(y+8·t4,t1,t2)
	ShiDian nao [13]	E(x,y | t1,· · · ,t6)→ P(t4,x+8·t5,y+8·t6)	E(x,y | t1,· · · ,t4)→ P(t1,x+8·t3,y+8·t4)
	Rubick	E(x,0 | t1,· · · ,t6)→ P(-x+t1, t5, x+8·t6)	E(x,0 | t1,· · · ,t4)→ P(-x+t1, t3, x+8·t4)
CONV+ FC [28]	NVDLA [47]	E(x,0 | t1,· · · ,t7)→ P(t6,x+8·t7,t1,t2)	E(0,y | t1,· · · ,t5)→ P(t1,y+8·t5,t2,t3)
	TPU [23]	E(0,y | t1,· · · ,t7)→ P(y,-y+t1,t6,t7)	E(x,0 | t1,· · · ,t5)→ P(-x+t1,x+8·t3,t4,t5)
	Rubick	E(0,y | t1,· · · ,t7)→ P(y,-y+t1,t6,t7)	E(0,y | t1,· · · ,t5)→ P(-y+t1,y+8·t3,t4,t5)
GEMM+ SoftMax+ GEMM [34]	TPU [23]	E(0,y | t1,t2,t3)→ P(-y+t1,y+8·t2)	E(x,0 | 0,t2,t3)→ P(-x+t1,x+8·t3)
	Rubick	E(x,y | 0,t2,t3)→ P(x+8·t3,y+8·t2)	E(x,y | 0,t2,t3)→ P(x+8·t3,y+8·t2)

dramatically because we separately form the sub-space of each IR and then compose them together. The initial Rubick space is 6,773,760 (196 points in access entry space, 34560 points in data layout space). The three pruning strategies furthers prunes the space. Clearly, pruning strategies 1 and 3 achieve 4.32X and 20.5X space reduction by pruning non-full-rank cases, respectively; pruning strategy 2 leads to 3.62X reduction by removing the wrong output cases.

Mathematically, each dataflow in TENET space can be uniquely decomposed into access entry and data layout. While the inferior dataflows involve inefficient data layouts that underutilize the PE array.

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

For example, the above dataflow leads to diagonal-interleaved PE utilization as shown in Figure 11. According to our dataflow decomposition methodology, such under-utilization results from inferior data layout that has fractional coefficiency, as follows.

$$L_{E_{st} \to D_A} = \{(E(x,0) \mid T(t)) \to A(0.5x - 0.5t, 0.5x + 0.5t)\}$$

However, Rubick effectively prunes these cases when forming the data layout IR space.

F. Multi-kernel Implementation on ASIC

Real-world tensor applications often involve multiple dependent kernels. For example, 3×3 CONV layers followed by 1×1 CONV layers are widely used in convolutional neural networks (CNNs) such as ResNet [35] and GoogleNet [65]. Prior accelerator designs usually process these kernels sequentially and consecutive layers use different data layouts for output and input. This leads to a large buffer or DRAM to cache the intermediate results. The architectural details exposed by

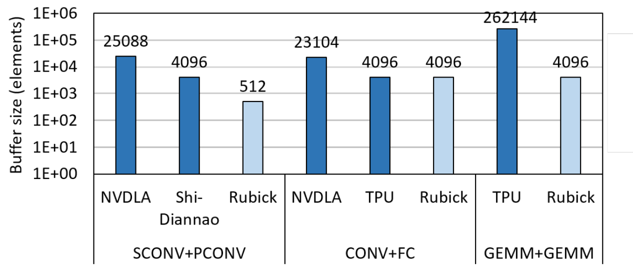


Fig. 12. Data layout IR optimization for multi-kernel.

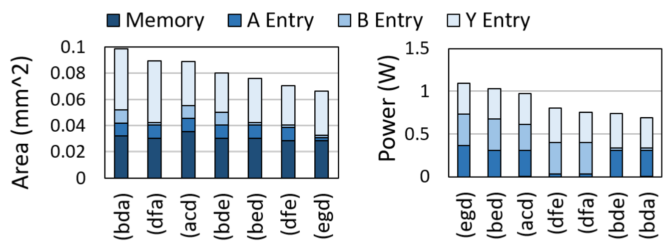


Fig. 13. Area and power breakdown via GEMM dataflow decomposition. The X-axis is different dataflows notated by access entry. E.g., **(bda)** means, tensor A, B, Y applies type-(a), type-(d), type-(b) access entry of Figure 4.

Rubick allow us to optimize the buffer size by using similar data layout IRs for multi-kernel dataflows.

Figure 12 compares buffer sizes, where SCONV+PCONV representing spatial convolution and point-wise convolution, respectively. Rubick achieves 49X and 8X reduction compared to NVDLA and ShiDiannao for the first case, respectively. It achieves 5.6X reduction compared to NVDLA for the second case and 64X reduction compared to TPU for the last case. Table II provides the analysis results using Rubick data layout IR. Shi-Diannao [13] maps the same tensor dimension to different time-stamps. In the output layout, the first tensor dimension is mapped to an outer time-stamp $t4$, while in the input layout, it is mapped to the innermost time-stamp $t1$. As a result, the tensor data in the first dimension needs to be buffered across multiple iterations, leading to large intermediate buffer size. Though TPU [23] shows the same buffer size as Rubick in CONV+FC benchmark, it requires a transposition operation for data rearrangement due to the different space-stamps between their output and input layout, with one indexed by $E(x, 0|\cdots)$ and the other $E(0, y|\cdots)$. Thus, it needs to reload the tile, resulting in the loss of the benefit gained from fusing two kernels, which leads to an increase in both data movement power consumption and computation time.

G. Area and Power Analysis on ASIC

In this section, we generate the design from our hardware generator and synthesize the RTL code to evaluate the trade-off between area and power. Figure 13 (a) presents the area breakdown of various GEMM dataflows on an 8×8 PE array with 16-bit integer arithmetic. The memory area is obtained from UMC 55nm SRAM library. We consider the minimum memory size that only stores the data required in the first time-stamp of data layout IR. 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 type-(e), type-(f), or type-(g) (refer to Figure 4) require less area as they only need wires to broadcast data. While systolic entries are implemented using FIFOs with control logic. In Figure 13 (b), the memory power is negligible due to the small PE array size. We observe that 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.

Based on the pre-synthesized results of each IR, Rubick can accurately estimate the area and power of a complete design. We estimate area and power by separately synthesizing the modules implementing each kind of access entry and adding them together. The golden result is acquired by synthesizing the complete design. Compared to the golden synthesis results, Rubick achieves an accuracy of 91.96% and 91.09% for area and power. For TPU [23] dataflow, Rubick is 91.27% and 92.69% accurate for area and power, while the accuracy of TENET is only 60.22% and 88.79%. This is because TENET relies on simple polynomials of its high-level metrics (Reuse Volume) to estimates area and power, while Rubick relies on low-level IRs with accurate architectural details.

TABLE III
FPGA Performance Comparison

	Device	LUT	DSP	BRAM	MHz	GFLOP/s
AutoSA [68]	U250	56%	77%	30%	272	950
TensorLib [21]	VU9P	73%	75%	73%	245	626
EMS-WS [22]	VU9P	76%	73%	53%	301	731
EMS-OS [22]	VU9P	83%	73%	53%	295	717
Rubick	VU9P	46%	70%	11%	333	1066

H. FPGA Implementation

Table III compares the FPGA performance of Rubick with AutoSA [68], TensorLib [21] and EMS [22] on 2D-CONV. We select the late layers on VGG-19 [61] 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 10.6%. 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 improved the peak performance by 12% and 49%, compared against AutoSA [68] and EMS-WS [22], respectively.

VII. Related Works

Dataflow Modeling. Dataflow modeling can provide general guidelines and insights for optimizing the dataflow. Prior dataflow models mainly focus on tensor applications on spatial architectures [8], [11], [20], [29], [30], [36], [39], [40], [49], [52], [71], [72]. A few of them propose dataflow notations to precisely describe how the instance is executed [11], [20],

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

[29], [30], [39], [49], [71]. For example, TENET [39] proposes relation-centric notation that regards the dataflow as a mapping function between iteration domain and hardware. In [11], dataflow is annotated using two hyperplanes with the polyhedral dependency graph. Kwon et al. [29], [30] propose a data-centric notation to specify the data distribution in spatial dimensions and time dimensions. Timeloop [49] and Interstellar [71] annotates dataflow using loop nest with some hardware directives. For example, Timeloop [49] introduces mapping directives for memory hierarchy and PE workload assignment. While Interstellar [71] extends Halide [58] with additional control directives, e.g., `loop blocking` and `resource allocation`, for specifying the hardware features. CoSA [20] uses a binary matrix to represent the spatial and temporal mapping. However it only aims at DNN and the solver-based approach does not support varied dataflows that that apply linear transformation between different dimensions. There are also prior works aiming at modeling the spatial architecture for general applications [36], [46], [52].

Spatial Architecture Generation. Spatial architectures require extensive manual effort to design the hardware modules. Therefore, many recent works propose generation tools to automatically design the architecture [10], [15], [26], [33], [53], [60], [68], [69]. DSAGEN [69] is a framework that applies a hardware/software co-design approach for generating reconfigurable architectures. DSAGEN proposes a compilation flow with a design space exploration algorithm based on modular architecture components. Spatial [26] is a domain-specific language for spatial accelerators, which provides hardware-specific abstractions for control, memory, and design tuning. μIR is an intermediate representation for describing the micro-architecture of spatial accelerators [60]. It decouples the architecture from the algorithm and is translated to Chisel for hardware generation. These works act like black boxes that transform the dataflow into IRs to generate architecture, however, make it difficult for users to interpret the relationship between dataflow and architecture.

VIII. Conclusion

In this work, we propose an infrastructure for analyzing, exploring, and implementing the architecture of spatial dataflows. 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 an efficient exploration approach by separately forming the subspace of these two intermediate representations. Finally, Rubick enables various low-level implementation optimizations, and accelerates the DSE time of dataflows by up to 1.1×10^5X, saving the time from days to minutes.

Acknowledgements

This work was supported in part by the National Key R&D Program of China under Grant (No. 2022YFB4500401). This work was also supported in part by National Natural Science Foundation of China (NSFC) under grant No.62090021 and No. 61825205.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Symposium on Operating Systems Design and Implementation*, 2016.

[2] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *The Journal of Machine Learning Research*, 2014.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the Design Automation Conference*, 2012.

[4] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, 2007.

[5] P. Chatarasi, H. Kwon, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, "Marvel: A data-centric approach for mapping deep learning operators on spatial accelerators," *ACM Trans. Archit. Code Optim.*, 2022.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, 2014.

[7] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, 2016.

[8] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[9] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[10] J. Cong and J. Wang, "PolySA: Polyhedral-Based Systolic Array Auto-Compilation," in *Proceedings of the International Conference on Computer-Aided Design*, 2018.

[11] S. Dave, A. Shrivastava, Y. Kim, S. Avancha, and K. Lee, "dMazeRunner: Optimizing Convolutions on Dataflow Accelerators," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2020.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.

[13] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, 2015.

[14] D. M. Dunlavy, T. G. Kolda, and W. P. Kegelmeyer, "Multilinear algebra for analyzing data with multiple linkages," in *Graph algorithms in the language of linear algebra*, 2011.

[15] D. Durst, M. Feldman, D. Huff, D. Akeley, R. G. Daly, G. L. Bernstein, M. Patrignani, K. Fatahalian, and P. Hanrahan, "Type-directed scheduling of streaming accelerators," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2020.

[16] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*. IEEE, 2018.

[17] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, 2012.

[18] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 943–958.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[20] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzynek, T. Norell, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *48th ACM/IEEE Annual International Symposium on Computer Architecture*, 2021.

[21] L. Jia, Z. Luo, L. Lu, and Y. Liang, "Tensorlib: A spatial accelerator generation framework for tensor algebra," in *58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[22] L. Jia, Y. Wang, J. Leng, and Y. Liang, "EMS: efficient memory subsystem synthesis for spatial accelerators," in *Proceedings of DAC*, 2022.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[24] S. Kao and T. Krishna, "GAMMA: automating the HW mapping of DNN models on accelerators via genetic algorithm," in *IEEE/ACM International Conference On Computer Aided Design, ICCAD*, 2020.

[25] S. Kao, A. Parashar, P. Tsai, and T. Krishna, "Demystifying map space exploration for npus," in *IEEE International Symposium on Workload Characterization, IISWC 2022, Austin, TX, USA, November 6-8, 2022*, 2022.

[26] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis *et al.*, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 296–311.

[27] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, 2009.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, 2012.

[29] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[30] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, 2020.

[31] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ACM SIGPLAN Notices*, 2018.

[32] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 242–251.

[33] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang *et al.*, "SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs," in *International Conference on Computer Aided Design*, 2020.

[34] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *ACL*, 2020.

[35] R. Likamwa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: Analog convnet image sensor architecture for continuous mobile vision," *Acm Sigarch Computer Architecture News*, 2016.

[36] D. Liu, S. Yin, L. Liu, and S. Wei, "Polyhedral model based mapping optimization of loop nests for CGRAs," in *Proceedings of the 50th Annual Design Automation Conference*, 2013.

[37] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, 2015.

[38] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. IEEE, 2016.

[39] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, and Y. Liang, "Tenet: A framework for modeling tensor dataflow based on relation-centric notation," in *2021 ACM/IEEE 48rd Annual International Symposium on Computer Architecture*, 2021.

[40] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture*, 2017.

[41] Z. Luo, L. Lu, S. Zheng, J. Yin, J. Cong, Y. Liang, and J. Yin, "Rubick: A Synthesis Framework for Spatial Architectures via Dataflow Decomposition," in *60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[42] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *the conference on Recommender systems*, 2013.

[43] L. Mei, P. Houshmand, V. Jain, J. S. P. Giraldo, and M. Verhelst, "Zigzag: A memory-centric rapid DNN accelerator design space exploration framework," 2020. [Online]. Available: https://arxiv.org/abs/2007.11360

[44] NELL-2, http://frostt.io/tensors/.

[45] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017.

[46] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *ACM SIGPLAN Notices*, 2013.

[47] NVIDIA, http://nvdla.org/, 2020.

[48] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[49] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software*, 2019.

[50] A. Parashar, P. Chatarasi, and P.-A. Tsai, "Hardware abstractions for targeting eddo architectures with the polyhedral model," in *11th International Workshop on Polyhedral Compilation Techniques (Vitural Event)(IMPACT 2021)*, 2021.

[51] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel *et al.*, "Triggered instructions: a control paradigm for spatially-programmed architectures," *ACM SIGARCH Computer Architecture News*, 2013.

[52] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[53] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 137–151.

[54] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017.

[55] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[56] X. Qingcheng, Z. Size, W. Bingzhe, X. Pengcheng, Q. Xuehai, and L. Yun, "HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2021.

[57] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[58] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, 2013.

[59] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[60] A. Sharifian, R. Hojabr, N. Rahimi, S. Liu, A. Guha, T. Nowatzki, and A. Shriraman, "μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 940–953.

[61] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *arXiv preprint arXiv:1409.1556*, 2014.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2023.3337208

[62] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the Workshop on Irregular Applications: Architectures and Algorithms*, 2015.

[63] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen *et al.*, "T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 181–189.

[64] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[65] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[66] S. Venkataramani, V. Srinivasan, W. Wang, S. Sen, J. Zhang, A. Agrawal, M. Kar, S. Jain, A. Mannari, H. Tran, Y. Li, E. Ogawa, K. Ishizaki, H. Inoue, M. Schaal, M. J. Serrano, J. Choi, X. Sun, N. Wang, C. Chen, A. Allain, J. Bonanno, N. Cao, R. Casatuta, M. Cohen, B. M. Fleischer, M. Guillorn, H. Haynie, J. Jung, M. Kang, K. Kim, S. Koswatta, S. K. Lee, M. Lutz, S. Mueller, J. Oh, A. Ranjan, Z. Ren, S. Rider, K. Schelm, M. Scheuermann, J. Silberman, J. Yang, V. Zalani, X. Zhang, C. Zhou, M. M. Ziegler, V. Shah, M. Ohara, P. Lu, B. W. Curran, S. Shukla, L. Chang, and K. Gopalakrishnan, "Rapid: AI accelerator for ultra-low precision training and inference," in *48th ACM/IEEE Annual International Symposium on Computer Architecture*, 2021.

[67] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2nd ACM workshop on Online social networks*, 2009.

[68] J. Wang, L. Guo, and J. Cong, "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA," in *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2021.

[69] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 268–281.

[70] Y. N. Wu, P. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical approach to sparse tensor accelerator modeling," in *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*, 2022.

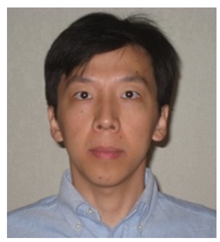
[71] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[72] D. Zhang, S. Huda, E. M. Songhori, K. Prabhu, Q. V. Le, A. Goldie, and A. Mirhoseini, "A full-stack search technique for domain optimized deep learning accelerators," in *Proceedings of ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[73] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

Liqiang Lu is a ZJU100 Young Professor in the College of Computer Science, Zhejiang University (ZJU), China. His research interests include quantum computing, computer architecture, deep learning accelerator, and software-hardware codesign. He has authored more than 20 scientific publications in premier international journals and conferences in related domains, including ISCA, MICRO, HPCA, ASPLOS, FCCM, DAC, IEEE Micro, and TCAD. He also serves as a TPC member in the premier conferences in the related domain, including ICCAD, FPT, HPCC etc.

Zizhang Luo is a PhD student in the School of Integrated Circuits, Peking University, China. His research interest includes computer architecture, hardware-software codesign, and electronic design automation. He has authored 4 scientific publications in top international conferences in the related domains, including DAC, ISCA and MICRO.



Yun (Eric) Liang is an associate professor (with tenure) in the School of EECS, Peking University, China. His research interests include computer architecture, compiler, electronic design automation, and embedded system. He has authored over 90 scientific publications in premier international journals and conferences in related domains. His research has been recognized by best paper awards at FCCM 2011 and ICCAD 2017 and best paper nominations at PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008. He serves as Associate Editor for ACM Transactions in Embedded Computing Systems (TECS), ACM Transactions on Reconfigurable Technology and Systems (TRETS), and Embedded System Letters (ESL). He also serves in the program committees in the premier conferences in the related domain including MICRO, DAC, HPCA, FPGA, ICCAD, FCCM, ICS, etc.

Jianwei Yin received the PhD degree in computer science from Zhejiang University (ZJU) in 2001. He was a Visiting Scholar with the Georgia Institute of Technology. He is currently a Full Professor with the College of Computer Science, ZJU. He has published more than 100 papers in top international journals and conferences. His current research interests include quantum computing, service computing and business process management. He is an Associate Editor of the IEEE Transactions on Services Computing.