Spatial Accelerator Generation and Optimization for Tensor Applications

by

Zhekai Zhang

B.Eng., Shanghai Jiao Tong University (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

Authored by: Zhekai Zhang
Department of Electrical Engineering and Computer Science
August 31, 2023

Certified by: Song Han
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Spatial Accelerator Generation and Optimization for Tensor Applications

by

Zhekai Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2023, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Modern foundation models and generative AI applications require multiple input modalities (both vision and language), which increases the demand for flexible accelerator architecture.

Existing frameworks suffer from the trade-off between design flexibility and productivity of RTL generation: either limited to very few hand-written templates or cannot automatically generate the RTL.

To address this challenge, we propose the LEGO framework, which automatically generates and optimizes spatial architecture design in the front end and outputs synthesizable RTL code in the back end without RTL templates. LEGO front end finds all possible interconnections between function units and determines the memory system shape by solving the integer linear equations, and establishes the connections by a minimum-spanning-tree-based algorithm and a breadth-first-search-based heuristic algorithm for merging different spatial dataflow designs. LEGO back end then translates the hardware in a primitive-level graph to perform lower-level optimizations, and applies a set of linear-programming algorithms to optimally insert pipeline registers and reduce the overhead of unused logic when switching spatial dataflows.

Our evaluation demonstrates that LEGO can achieve 3.2× speedup and 2.4× energy efficiency compared to previous work Gemmini, and can generate one architecture for diverse modern foundation models in generative AI applications.

Thesis Supervisor: Song Han
Title: Associate Professor of Electrical Engineering and Computer Science

Disclaimer

Part of the work in this thesis was done in collaboration with another student, Yujun Lin. The credit for designing the high-level framework and representation is shared by both of us. Besides that, my work focuses mainly on the design and implementation of LEGO back end, as well as the evaluation of generated hardware.

Acknowledgments

First of all, I would like to express my deepest gratitude to my advisor, Professor Song Han. This thesis would not have been possible without his support and guidance. I have greatly benefited from his constructive feedback during our discussions and I learned a lot about how to conduct impactful research. It is my fortune to have Professor Han as my advisor for research.

I would also like to thank my undergraduate advisors, Dr. Chao Li and Professor Yong Yu, for introducing me to the research of computer architecture, and Professor Vivienne Sze, Professor Joel S. Emer, Professor William J. Dally, for guiding me to the field of domain-specific architecture and AI accelerators.

Meanwhile, I would deliver my appreciation to my collaborator, Yujun Lin. He proposed many awesome ideas when designing LEGO and helped me a lot during the implementation and debugging.

Besides, I also thank Hanrui Wang, for inspiring discussions about NLP models and hardware, Ji Lin and Jiaming Tang, for great ideas about FP4 quantization, and all colleagues at MIT HAN LAB, for impressive brainstorms about efficient AI.

Part of this research is funded by MIT-IBM Watson AI Lab, MIT AI Hardware Program, Amazon and MIT Science Hub, NVIDIA Academic Partnership Award and NSF. I'm sincerely grateful for their generous support.

Finally, thanks to my father, Xu Zhang, and my mother Suqin Shen, for their unwavering support and love. Their encouragement and belief in me have been a constant source of inspiration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

The proliferation of tensor applications, particularly deep neural networks, has led to an unprecedented demand for efficient and high-performing solutions. Especially in the era of multi-modal large foundation models, a single accelerator needs to process/generate both language and vision inputs/outputs [8, 22, 32], such as GPT-4 [23] and stable diffusion [11]. Many spatial hardware architectures have been proposed to accelerate common operations, such as TPU [13] for general matrix multiplication (GEMM), Eyeriss [1] for convolutional neural networks, and A3 [9], SpAtten [35], and Sanger [20] for attention operation in transformers [33], and PointAcc [18] for sparse convolution in point cloud deep learning [2]. Although these architectures have exhibited remarkable performance for the target operations, their development cycle is frequently impeded by extensive human effort. This drawback puts them at a disadvantage when handling emerging algorithms that require diverse new operations. Moreover, these human-designed accelerators cannot cover the entire design space of architectures, making them sub-optimal for certain situations. Therefore, an automatic hardware design methodology is urgently needed to reduce human effort and shorten the development cycle for highly flexible architectures.

Another benefit of automatic hardware design tools is the fair and end-to-end evaluation of emerging techniques. For example, with the rapid development of large

language models, many techniques are proposed to reduce the memory bandwidth bottleneck by using low-bitwidth quantization, like VS-Quant [3] and microexponent [5]. However, it is difficult to directly compare the results of these works due to the differences in experiment settings. An automatic hardware design tool can work as a platform to evaluate them by generating the end-to-end power/performance/area results under the same setting with little extra human effort, which can further speed up the development of new techniques.

In support of spatial architecture development for tensor applications, dataflow analysis tools like Timeloop [24], MAESTRO [15], and TENET [19] have been proposed to model the hardware behavior and predict their latency and energy. Recently, design space exploration tools such as NAAS [17] and MAGNET [34] have leveraged these tools to search for optimal architectures in the design space.

However, one major obstacle is the absence of powerful generation tools that can effectively translate any hardware abstractions in the analysis tools into actual RTL code. This is particularly challenging given the increasing diversity of tensor operations in modern models, which require flexible dataflows in one accelerator to achieve better performance. However, as shown in Table 1.1, current generation tools [7, 28, 34, 40] heavily rely on pre-defined handwritten templates, which can be too rigid and severely limit the hardware design space, ultimately impeding their practical usefulness.

Another major obstacle is the lack of a comprehensive representation of the hardware. Current solutions [7, 17, 34, 39] typically use template hyperparameter vectors to describe the hardware design. However, such abstraction does not capture the intricate connectivity of different hardware components, resulting in limited hardware design space and little assistance in automatic hardware generation.

1.2 Framework Overview

To address these challenges, we propose a novel end-to-end framework, LEGO, as shown in Figure 1-1. Unlike traditional methods that rely on handwritten RTL/HLS

Table 1.1: Comparison between Different Spatial Accelerator Generators and Analysis Frameworks

Work	**RTL Generation**	**Design Space**
Timeloop [24], MAESTRO [15]	✗	unlimited
TENET [19]	✗	unlimited
Gemmini [7], DNNWeaver [28]	✓ (Template-based)	limited to 2 templates
MAGNET [34]	✓ (Template-based)	limited to 1 template
DNA [40]	✓ (Template-based)	limited to 3 dataflows
AutoSA [36]	✓ (HLS for FPGA)	limited to 2D systolic array
Lego (ours)	✓	unlimited

templates, LEGO directly generates spatial architecture design (via front end) and outputs synthesizable RTL code (via back end) without sacrificing flexibility from a comprehensive high-level hardware abstraction, as shown in Figure 1-2.

Specifically, to enable full design freedom on interconnections topology, LEGO builds hardware based on a hierarchical spatial architecture paradigm that separates computation logic, data path topology, and memory system for each parallelism level. LEGO also adopts relation-centric high-level representation that focuses on the mappings between the iteration domain and (par)for-loop instances in the workload.

LEGO front end captures the data behavior in the target workload and determines the connectivity between different functional units (FUs) and memory allocation by directly solving the linear equations. The front end formulates the connectivity decision into a graph optimization problem, exploits a Minimum Spanning Tree (MST) based algorithm to maximize the data reuse, and employs a Breath First Search (BFS) based heuristic algorithm that merges connections for various dataflows to minimize the data path overhead. The front end then generates an FU-level architecture description graph.

LEGO back end further translates the graph into a primitive-level detailed architecture graph (DAG), of which the basic components are hardware primitives such as multipliers, adders, and buffers. To help the design meet the timing requirement, the

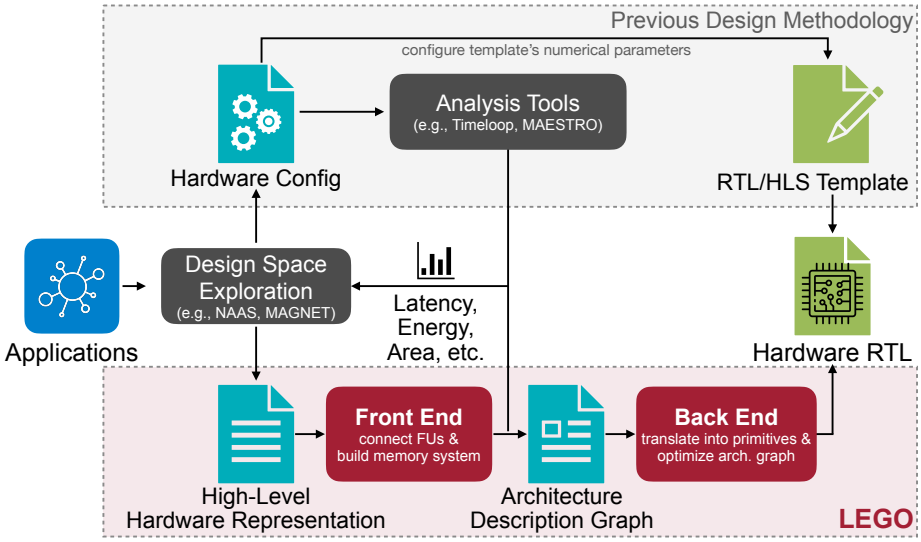

Figure 1-1: Instead of configuring the sizing parameters in the hardware template, LEGO directly generates spatial architecture design and outputs RTL code from high-level hardware description.

back end applies a set of linear-programming algorithms to optimally insert pipeline registers with minimum cost. To further reduce area and power overhead, The back end performs a pin-reusing algorithm to eliminate the overhead of unused computation logic and deploys a power-gating logic for unused connections when switching different dataflows in the hardware.

1.3 Thesis Overview and Contributions

This thesis introduces the whole LEGO framework following Figure 1-2.

Chapter 2 provides a general overview of the spatial architecture generated by LEGO and introduces the basic modules and the memory hierarchy in the hardware.

Chapter 3 introduces the high-level representation used in LEGO framework, which describes mappings between iteration, temporal for-loop instances, spatial FU arrays, and operand tensor data. LEGO front end will take advantage of these mappings to analyze data reuse and build interconnections.

Chapter 4 describes the design of LEGO front end. LEGO front end builds the architecture description graph and applies multiple algorithms, including MST and a

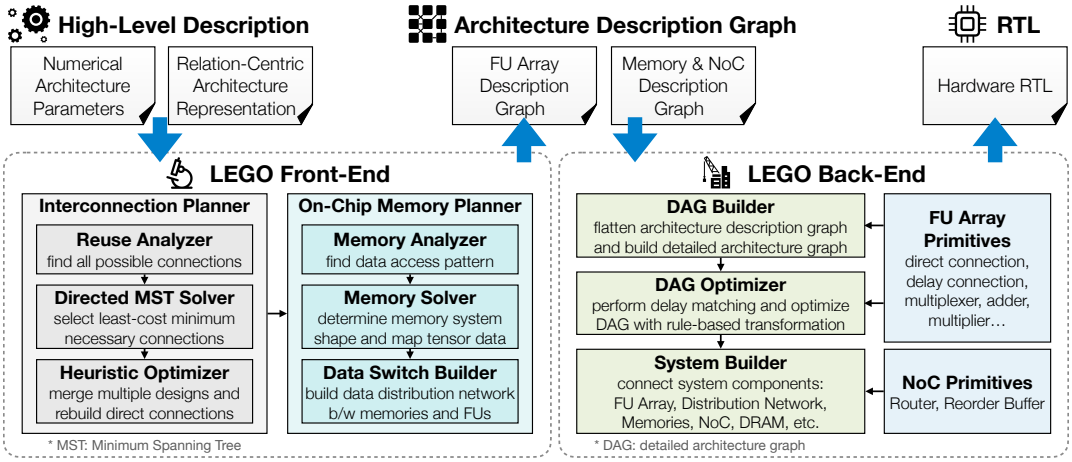

Figure 1-2: Overview of LEGO Spatial Accelerator Generator and Optimization Framework: from high-level description to hardware RTL.

BFS-based heuristic algorithm to optimize the overhead of the data path.

Chapter 5 describes the design of LEGO back end, including a detailed architecture graph to efficiently model the low-level hardware components, and multiple transformation passes to reduce the register cost and energy consumption.

Chapter 6 evaluates the LEGO framework using various benchmarks from single operations like GEMM, to complex tasks like neural networks, and further extends to diverse generative AI applications including the LLaMA foundation model and stable diffusion model. The evaluation results show that LEGO outperforms Gemmini [7] with 3.2× speedup and 2.4× energy efficiency. We also provided an example of using LEGO as an end-to-end evaluation tool for emerging quantization techniques.

1.4 Related Work

DSAGen [37] is a CGRA-based architecture that targets at high reconfiguability. Compared to DSAGen, our proposed LEGO framework puts more effort into performance of real-world applications. LEGO emphasizes data reuse (e.g. reuse analysis on relation-centric representation) and distribution (e.g. MST optimizer) and minimizes the cost of the data path, while DSAGen uses flexible switches to distribute data. LEGO uses human-designed transformation passes which can get near-optimal

hardware cost with polynomial time complexity based on the presumptions, while DSAGen relies on random search which is more suitable for unpredictable situations. LEGO has comparable efficiency as human designs (Table 6.2) while DSAGen has >2x area and power overhead (compared to DianNao). However, DSAGen natively supports sparse operation but LEGO needs special design.

Polyhedral Model is a loop-analyzing model similar to our relation-centric representation. However, there are still some differences between them. The polyhedral model targets loop optimization, while our representation aims more at hardware generation. We separate the spatial/temporal dataflows, and introduce control flow to constrain data dependency among FUs. Contrary to the polyhedral model's relation direction, we map from dataflow to the iteration domain to eliminate modulo operations in the representation. All relations are represented through affine transformation, simplifying interconnection analysis as solving integer equations.

Design Space Exploration Tools While not targeting generating actual hardware, there are many design space exploration tools proposed to estimate and optimize the cost of neural network accelerators. Examples include compute-centric Timeloop [24], data-centric MAESTRO [15], relation-centric TENET [19], and optimization tools like NAAS [17]. These tools tend to do some simplification to the hardware (for example, by ignoring the bank of buffers and detailed FU connections) and perform the first-order estimation of the hardware performance. On the contrary, LEGO accurately models the hardware to achieve end-to-end RTL generation. However, these tools can provide good feedback when choosing the high-level parameters for LEGO and can work together to create an end-to-end solution of software-hardware co-design.

Chapter 2

LEGO Spatial Architecture Overview

As shown in Figure 2-1, LEGO spatial architecture provides flexibility and high expressiveness by decoupling computation units, data paths, and memory systems (Figure 2-1(a)) on a hierarchical design (Figure 2-1(b)). The architecture consists of several basic modules, including:

Functional Units (FUs) are the smallest indecomposable building blocks of computation. It executes the computation defined in the loop body. For example, the FU of GEMM and convolution workload is multiplication-add (MAC): $Y+ = A \cdot B$; the FU of the mixed-precision GEMM workload using BitFusion [29] architecture is the 2-bit multiplication-and-shift-add unit: $Y = Y + (A \cdot B) << C$. LEGO does not generate the FU microarchitecture but rather employs a user-defined FU design to offer customization.

FU Interconnections specify the direct communication between functional units. There are two types of FU interconnections: direct and delay interconnection. Delay interconnections are essentially FIFOs parameterized by their depth. A deeper FIFO implies more area but allows larger loop tiling sizes. The depth of FIFO is programmable during runtime, offering a configurable delay when sharing the data. Different tensors have their own set of FU interconnections. The interconnections allow arbitrary but pre-defined data transfer between FUs. While there might be

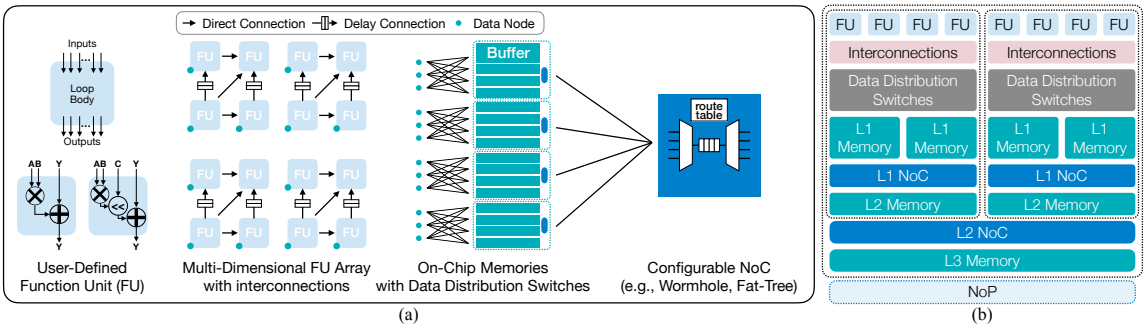

Figure 2-1: LEGO Spatial Architecture decouples computation logic, data path topology, and memory system in a hierarchical design.

multiple paths between two FUs, in each valid data flow configuration, the interconnection will form a directed forest and result in a fixed direction of data transfer. The allows a simple design of software (no complex scheduling required) and prevents any potential deadlock (since the forest must be acyclic).

Data Distribution Switches route the data between on-chip memories and functional units. For example, it can be a chubby tree in MAERI [16] and multi-cast in Eyeriss [1]. As the data reuse between FUs is handled by FU interconnections, data distribution switches resolve the memory access conflicts arising from the tensor data layout in on-chip memories.

Memories are parameterized by their capacity and width. L1 memories are assigned to different tensors. Each L1 memory space has only one address generator and controller. L2 and higher-level memories are shared by all tensors and use a Buffet-based interface to buffer the transmitted data.

Network-on-Chip (NoC) connects different memories at the same level and interacts with the memories at the higher level. Since data layout in L1 memories is determined by the data access pattern in FU array while data layout in higher-level memories is in line with the layout in off-chip memories, L1 NoC supports strided memory access and tensor transpose in addition to communication. We used a wormhole-based structure for the L2 NoC, and the deadlock is prevented using a

classical X-Y routing method.

Post-processing units (PPUs) process the layers that cannot trivially map to the FU array in neural network workloads. These layers include activation functions like ReLU, sigmoid, softmax, and GELU, and normalization layers like LayerNorm and GroupNorm. Each post-processing unit consists of a lookup table to calculate the activation function and a reduction unit to summarize the mean/variance during the normalization or sum of exponents during softmax. The post-processing units share the output buffers with the FU array to support in-place activation calculations and minimize hardware overhead during normalization.

Chapter 3

LEGO High-Level Representation

Relation-centric high-level representation in LEGO essentially describes mappings among different domains, including iteration domain, temporal for-loop instances, spatial FU arrays, and operand tensor data domains. By analyzing these mappings, we can derive the patterns of when a tensor element is accessed in the FU and how it is reused across FUs, and finally, determine the interconnections between FUs and the allocation of tensor data on memories.

3.1 Workload Representation

A tensor workload is essentially the mappings between its iteration domain and tensor domains.

Definition 1: **Data Mapping.** Given a n_I-dimensional iteration domain I and a n_D-dimensional tensor domain D, a data mapping relation is defined as,

$$\vec{d} = \mathbf{M}_{I \to D} \vec{i} + \mathbf{b}_{I \to D}, \tag{3.1}$$

where $\vec{d}$ is a n_D-d vector indicating a tensor D element, $\vec{i}$ is a n_I-d vector indicating a iteration instance in I, and $\mathbf{M}_{I \to D}$ is a affine transformation matrix with a size of $n_D \times n_I$, $\mathbf{b}_{I \to D}$ is a n_D-d bias.

In the example of Figure 3-2 (the 2D-Conv workload $\mathbf{Y}_{n,oc,oh,ow} = Conv\left(\mathbf{X}_{n,ic,ih,iw}, \mathbf{W}_{oc,ic,kh,kw}\right)$),

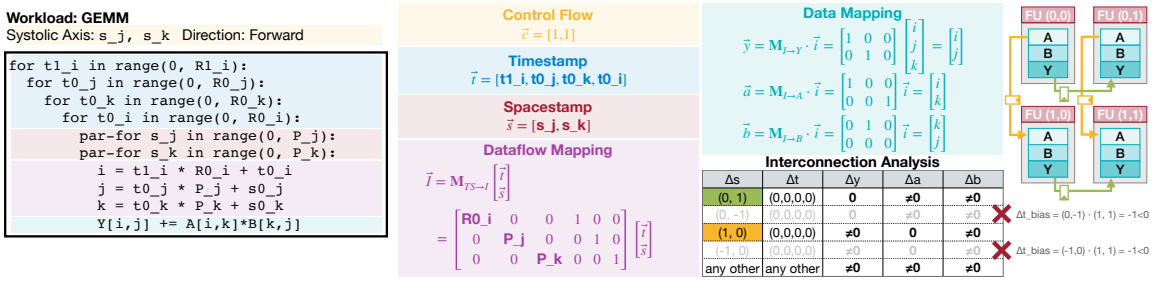

Figure 3-1: GEMM workload example of relation-centric notation and interconnection analysis

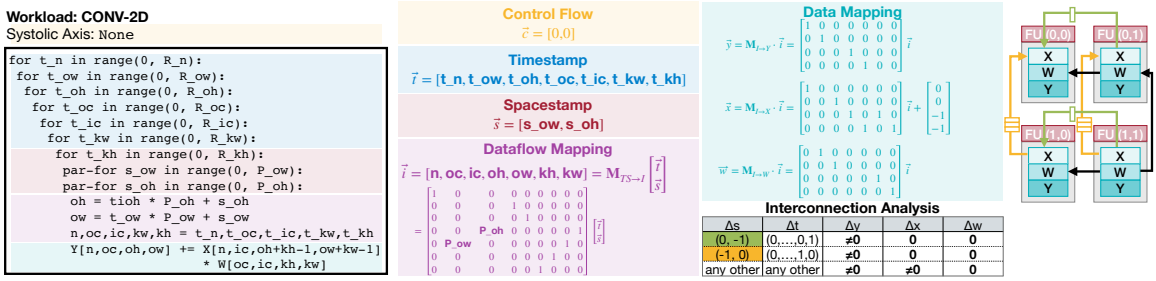

Figure 3-2: 2D convolution example of relation-centric notation and interconnection analysis

the data mapping of input tensor X can be represented as,

$$\vec{x} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} n \\ oc \\ ic \\ oh \\ ow \\ kh \\ kw \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \\ -1 \end{bmatrix}$$

where the iteration domain of 2D-Conv has 7 dimensions: N (batch size), OC/IC (output/input channels), OH/OW (output height/width), and KH/KW (kernel height/width).

3.2 Dataflow Representation

Dataflow representation contains temporal dataflow and spatial dataflow.

Definition 2: **Dataflow Mapping**. Given a n_I-dimensional iteration domain I, n_T-level for-loop nests T, n_S-level parfor-loop nests S, the dataflow mapping is defined

as,

$$\vec{i^S} = \mathbf{M}_{S\to I}\vec{s}, \tag{3.2}$$

$$\vec{i^T} = \mathbf{M}_{T\to I}\vec{t}, \tag{3.3}$$

$$\vec{i} = \vec{i^T} + \vec{i^S} = \mathbf{M}_{T,S\to I}\begin{bmatrix}\vec{t}\\ \vec{s}\end{bmatrix} = \begin{bmatrix}\mathbf{M}_{T\to I} & \mathbf{M}_{S\to I}\end{bmatrix}\begin{bmatrix}\vec{t}\\ \vec{s}\end{bmatrix} \tag{3.4}$$

$\vec{s}$ is a n_S-d vector indicating the FU coordinates where the loop instance is executed. $\vec{t}$ is a n_T-d vector indicating the coordinates of the for-loop state. The lexicographical order of $\vec{t}$ indicates the for-loop execution order. That is, the first dimension of $\vec{t}$ is the outermost for-loop and the last dimension of $\vec{t}$ is the innermost for-loop. Therefore, given the for-loop sizes $\vec{R_T}$, we can convert $\vec{t}$ into a scalar integer t as follows,

$$t = \begin{bmatrix}0 & \vec{t}^{\,T}\end{bmatrix}\begin{bmatrix}\vec{R_T}\\ 1\end{bmatrix} = ((t_0 \cdot R_{T_1} + t_1)\cdot R_{T_2} + t_2)\cdots \tag{3.5}$$

where $\vec{t} = [t_0, t_1, \cdots]^T$ and $\vec{R_T} = [R_{T_0}, R_{T_1}, R_{T_2}, \cdots]^T$. $\mathbf{M}_{T\to I}$ with size of $n_I \times n_T$ and $\mathbf{M}_{S\to I}$ with size of $n_I \times n_S$ are both non-negative affine transformation matrices. They can all be inferred from the for-loop sizes $\vec{R_T}$ and parfor-loop sizes $\vec{R_S}$ (*i.e.*, the FU array sizes), and guarantee that all relations in Equations 3.2,3.3, 3.4 are bijective. Note that for both $\mathbf{M}_{T\to I}$ and $\mathbf{M}_{S\to I}$, there is only one nonzero element per column; that is, $\mathbf{M}_{T\to I}$ only has n_T nonzeros and $\mathbf{M}_{S\to I}$ only has n_S nonzeros.

By manipulating the transformation matrix $\mathbf{M}_{S\to I}$ and $\mathbf{M}_{T\to I}$, our representation is able to describe diverse dataflows and support arbitrary loop tiling and loop reordering. Tiling a level of for-loop is equivalent to adding another dimension in T, *i.e.*, n_T is increased by 1. Note that n_T should be no larger than the number of loop counters in hardware, and no smaller than $\min(n_I - n_S, 0)$ to cover the whole iteration space I.

Figure 3-1(purple region) demonstrates an example of translating loop nests of GEMM into relation-centric dataflow mapping representation. The transformation matrix $\mathbf{M}_{S\to I}$ and $\mathbf{M}_{T\to I}$ only contain zeros and values from the loop sizes (*e.g.*,

R0_i, P_j). Furthermore, even though the iteration domain of GEMM has only $n_I = 3$ dimensions (I, J, K), the number of dimensions of timestamp $\vec{t}$ is $n_T = 4 > n_I$. Figure 3-2(purple region) shows another example of using relation-centric representation to express 2D convolution dataflow used in ShiDianNao architecture.

3.3 Control Flow and Systolic Array.

The control flow is denoted by a n_S-d ternary vector $\vec{c}$. Value one (or negative one) indicates the control signals flow in the forward (or backward) systolic manner along the corresponding dimension, while value zero means control signals are directly shared among the FUs in that dimension. As shown in the example in Figure 3-1, since control signals are forwarded along both spatial dimensions, the control flow is represented as $\vec{c} = [1, 1]$. While in the example in Figure 3-2, all FUs share the control signals at the same time and thus $\vec{c} = [0, 0]$.

Since the delay of control signals results in the delay of timestamp t, a timestamp bias t_{bias} can be assigned to each FU to represent such delay. t_{bias} can be calculated as,

$$t_{bias,\vec{s}} = \vec{s}^{\,T} \cdot \vec{c} \tag{3.6}$$

and the difference in timestamp $\vec{t}$ of any two FUs ($\vec{s}_1$ and $\vec{s}_2$) can be calculated as,

$$\Delta t_{bias,\vec{s}_1 \to \vec{s}_2} = t_{bias,\vec{s}_2} - t_{bias,\vec{s}_1} = (\vec{s}_2 - \vec{s}_1)^T \cdot \vec{c} = \Delta\vec{s} \cdot \vec{c} \tag{3.7}$$

Unlike TENET where systolic data sharing is represented using a relation from I to T (the reverse of our mapping), we treat the systolic manner of sharing tensor data as a natural result of the systolic manner of sharing control signals. As in the example of Figure 3-1, since control flow is $\vec{c} = [1, 1]$, multicast/reduction-tree is converted to systolic forward/reduction by adding a delay of $\Delta t_{bias} = 1$ on corresponding FU interconnections.

3.4 Interconnection Representation

Given a FU array, the FU Interconnections can be represented as a set of directed graphs, where each directed graph $G(\mathbf{V}, \mathbf{E}_D)$ corresponds to a certain tensor operand/result D (*e.g.*, input activation tensor, weight tensor). Each node in the graph is a FU, that is $\mathbf{V} = \{\vec{s} \mid \mathbf{0} \leq \vec{s} \leq \vec{R_S}\}$. Each edge is an interconnection represented with a tuple $\vec{e} = (\vec{s_1}, \vec{s_2}, \Delta t_{bias} + w)$, which indicating the data are transferred from FU $\vec{s_1}$ to $\vec{s_2}$ with a delay of $\Delta t_{bias} + w$ cycles. Usually, $\vec{e}$ is further constrained by $|\vec{s_1} - \vec{s_2}|_\infty \leq d_S, 0 \leq w \leq d_T$. That is, d_S constrains the maximum spatial distance of 'adjacent' FUs who can reuse their data *without* extra access from/to the scratchpad. d_T constrains the maximum depth of FIFOs that are inserted in interconnection $\vec{e}$ to hold the data for temporal reuse between FUs (*i.e.*, w is the programmable depth of the FIFO).

For example, the interconnections of a widely used 2D-systolic array with a size of 8×8 can be written as,

$$\mathbf{E}_A = \{\vec{e} = (\,[i,j],\ [i,j+1],\ 1\,) \,|\, 0 \leq i \leq 7, 0 \leq j \leq 6\}$$
$$\mathbf{E}_Y = \{\vec{e} = (\,[i,j],\ [i+1,j],\ 1\,) \,|\, 0 \leq i \leq 6, 0 \leq j \leq 7\}$$

In the example of Figure 3-2 (ShiDianNao architecture), the interconnections of input tensor X can be represented as,

$$\mathbf{E}_X = \{(\,[i,j+1],\ [i,j],\ w_0\,) \,|\, 0 \leq i \leq 7, 0 \leq j \leq 6, w_0 > 0\}$$
$$\cup \{(\,[i+1,j],\ [i,j],\ w_1\,) \,|\, 0 \leq i \leq 6, 0 \leq j \leq 7, w_1 > 0\}$$

Chapter 4

LEGO Front End

As shown in Figure 1-2, LEGO front end is designed to analyze the high-level representation and generates the architecture description graph. The target of LEGO front end is to build the interconnections between FUs with minimum cost. This is achieved by data reuse analysis and data path optimizations described below.

4.1 Relation-based Interconnection Analysis

Based on relation-centric representation, we can determine all possible interconnections between FUs by analyzing the tensor data reuse between FUs. That is, if there is a reuse of data between two FUs, there is a potential interconnection between these two FUs.

Direct Interconnection From Equations 3.1, 3.2 and 3.4, we find $\vec{\Delta s}$ that satisfies,

$$\begin{gathered} \mathbf{M}_{I\to D}\mathbf{M}_{S\to I}\vec{\Delta s} = \mathbf{0} \\ constraint: \left|\vec{\Delta s}\right|_{\infty} \leq d_S, \Delta t_{bias,\vec{\Delta s}} \geq 0 \end{gathered} \tag{4.1}$$

Each solution $\vec{\Delta s}$ can be converted into a set of interconnections as follows,

$$\mathbf{E}^{direct}_{\vec{\Delta s}} = \left\{ (\vec{s}, \vec{s}+\vec{\Delta s}, \Delta t_{bias,\vec{\Delta s}}) \mid \vec{s}, \vec{s}+\vec{\Delta s} \in \mathbf{V} \right\} \tag{4.2}$$

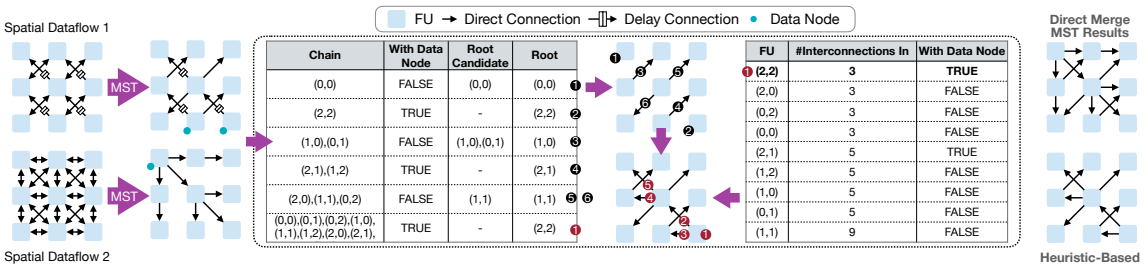

Figure 4-1: Heuristic-based Direct Interconnections Planning: 1. build interconnections for shorter chain first; 2. select FUs with input delay interconnections as possible root candidates of the chain; 3. select all FUs in the chain as possible root candidates if step 2 fails; 4. select FU with least #input direct interconnections and data node as final root; 5. extend the ongoing chain with a breadth-first search to connect the longest-built chains.

In the GEMM example in Figure 3-1, we find a solution $\vec{\Delta s} = (1, 0)$ for tensor A. Therefore, FU(0,0) pushes tensor A data to FU(1,0)=(0,0)+(1,0) and similarly FU(0,1) pushes A to FU(1,1). We also find a solution $\vec{\Delta s} = (0, 1)$ for tensor Y. Thus, FU(0,0) pushes tensor Y data to FU(0,1) = (0,0)+(0,1), and similarity FU(1,0) pushes Y to FU(1,1). Note that, in this example, even though $\mathbf{M}_{I\to D}\mathbf{M}_{S\to I}\vec{\Delta s} = \mathbf{0}$ for $\vec{\Delta s} = (-1, 0)$, it is impossible to establish an interconnection from FU(1,0) to FU(0,0)=(1,0)+(-1,0). This is because the difference of the temporal bias for this connection is $\vec{\Delta s}^{\,T}\vec{c} = -1 < 0$, which means the timestamp of FU(1,0) is always 1 cycle behind FU(0,0) and thus FU(1,0) cannot push tensor X data from future to past.

Delay Interconnection From Equations 3.1, 3.3 and 3.4, we solve the following equation,

$$\begin{aligned} \mathbf{M}_{I\to D}\mathbf{M}_{T\to I}\vec{t} &= -\mathbf{M}_{I\to D}\mathbf{M}_{S\to I}\vec{\Delta s} \neq \mathbf{0} \\ constraint: \left|\vec{\Delta s}\right|_\infty &\leq d_S, \Delta t_{bias,\vec{\Delta s}} \geq 0, \prod_{t_i\neq 0} t_i \leq d_T \end{aligned} \tag{4.3}$$

If expected hardware supports more than one type of temporal dataflow for a certain workload, we assume the for-loop nest domain is the same as the iteration domain $T = I$. Equation 4.3 is a classic Diophantine equation where $\vec{t}$ is the vector of

unknowns, which can be solved in polynomial time complexity.

Each solution $\vec{\Delta s}$ can be converted into a set of interconnections as follows,

$$\mathbf{E}^{delay}_{\vec{\Delta s}} = \left\{ (\vec{s}, \vec{s} + \vec{\Delta s}, w + \Delta t_{bias,\vec{\Delta s}}) \mid \vec{s}, \vec{s} + \vec{\Delta s} \in \mathbf{V}, w > 0 \right\} \tag{4.4}$$

If expected hardware only supports one type of temporal dataflow, we will set w to the minimum value t that satisfies Equation 4.3.

In the 2D convolution example in Figure 3-1, we get a solution pair $\vec{\Delta s} = (0, -1)$ and $\vec{t} = (0, \cdots, 0, 1)$ for tensor X. Therefore, FU(0,1) pushes tensor X data to FU(0,0)=(0,1)+(0,-1) and similarily FU(1,1) pushes X to FU(1,0). We also get a solution pair $\vec{\Delta s} = (-1, 0)$ and $\vec{t} = (0, \cdots, 1, 0)$ for tensor X. Therefore, in addition to FU(0,1), FU(1,0) also pushes tensor X data to FU(0,0)=(1,0)+(-1,0). Since both solutions are delay connections, the tensor X data received from FU(0,1) will be valid if and only if the timestamp of FU(0,0) is no smaller than $(0, \cdots, 0, 1)$, and data received from FU(1,0) will be valid if and only if the timestamp of FU(0,0) is no smaller than $(0, \cdots, 1, 0)$.

4.2 Minimum-Spanning Interconnections Generation

As depicted in the leftmost of Figure 4-1, FU interconnections found in the previous step may be excessive. Therefore, after obtaining all possible FU interconnections, we will search the minimum spanning trees on their union to get the minimum set of necessary connections for tensor data reuse:

$$\mathbf{E}^{MST} = \texttt{MinimumSpanningTrees}(\bigcup_{\Delta \vec{s}} \mathbf{E}_{\Delta \vec{s}}) \tag{4.5}$$

Since the entire graph is directed, we use Tarjan's Chu-Liu algorithm [31]. Note that for output interconnections, the direction of the edge will be reversed during the search since output buffers are data sinks instead of data sources. The root FU of every tree will be labeled with a data node indicating it requires fetching/committing the data from/to memories.

4.3 Heuristic-based Direct Interconnections Planning

If the expected hardware supports multiple spatial dataflows, we have to combine interconnections under different dataflows into one graph. However, directly merging all the minimum-spanning interconnections obtained from the previous step ($\bigcup_{\mathbf{M}_{S\to I}} \mathbf{E}^{MST}_{\mathbf{M}_{S\to I}}$) will not yield the optimal solution, especially for direct interconnections. Therefore, we propose a breadth-first search (BFS) based heuristic algorithm to re-establish all direct interconnections as illustrated in Figure 4-1.

First, we partition the FUs in the graph of each spatial dataflow ($\mathbf{M}_{S\to I}$): all FUs in the same subgraph can be connected to each other with direct interconnections. We refer to such subgraph as a *chain*. The FU providing the shared data is referred to as the *root* of the chain. The FUs receiving data with delay interconnections will be recorded as the root candidates. Each chain will be labeled if it requires data from memories. For example, in the spatial dataflow 1 of Figure 4-1, FU(2,0), FU(1,1) and FU(0,2) share the data, and thus they form a chain. Since only FU(1,1) pulls data through the delay interconnection, the root candidates of this chain only contain FU(1,1).

Then, we build direct interconnections for each chain one by one, following the order of the chain length. The root of the ongoing chain is selected from its root candidates: the FU having the least number of possible input direct interconnections will be selected. Starting from the root, we extend the ongoing chain with a breadth-first search to connect the longest-built chains. For example, when working on the longest chain in Figure 4-1, all FUs are in the root candidates. FU(2,2), FU(2,0), FU(0,2) and FU(0,0) only have 3 input direct interconnections (see the leftmost bottom graph). Furthermore, FU(2,2) is labeled with a data node in the previous shorter chains (the chain with black circled 2). Therefore, FU(2,2) will be selected as the final root of this longest chain. Among all neighbors of root FU(2,2), both FU(1,1) and FU(2,1) are the roots of the chain. Since the chain with FU(1,1) as root is longer, FU(2,2) will be first connected to FU(1,1) (red circled 2 in the figure) and then connected to FU(1,0). Moving from the root FU(2,2), we will then check the connectivity of

FU(1,1) and FU(2,1) similarly (red circled 4 and 5 in the figure).

After determining the direct interconnections in the previous step, we will add delay interconnections for the root FU in each direct interconnection chain.

Chapter 5

LEGO Back End

The architecture description graph (ADG) generated by LEGO front end provides the most important information about the hardware - the connections and memories. However, there is still a gap between the ADG and the actual RTL code: ADG describes the hardware at a relatively high level, of which the basic components are FUs, data nodes, and connections between them. However, they are not the elementary components in the hardware - a FU may perform different operations in different dataflows; a connection need to be implemented in FIFOs, wires, broadcast, or reduction tree; multiple connections to a single FU need to be merged by MUXes or reduction logic. ADG ignores the data types and bitwidths of signals, and uses an ideal hardware model: it assumes all of the computations are implemented in combinational logic and the result is ready immediately when data are read from buffers. This simplifies the front-end analysis but it is almost not realistic.

To get a more detailed view of the hardware, LEGO back end introduces a lower-level representation - detailed architecture graph (DAG). As illustrated in Figure 5-1, unlike ADG, which defines the connections between FUs, the DAG describes the hardware at the primitive level. Like any other compiler design, LEGO back-end first performs a translation pass (codegen) on ADG to build the DAG and then uses multiple transformation passes on the DAG to optimize and elaborate the hardware.

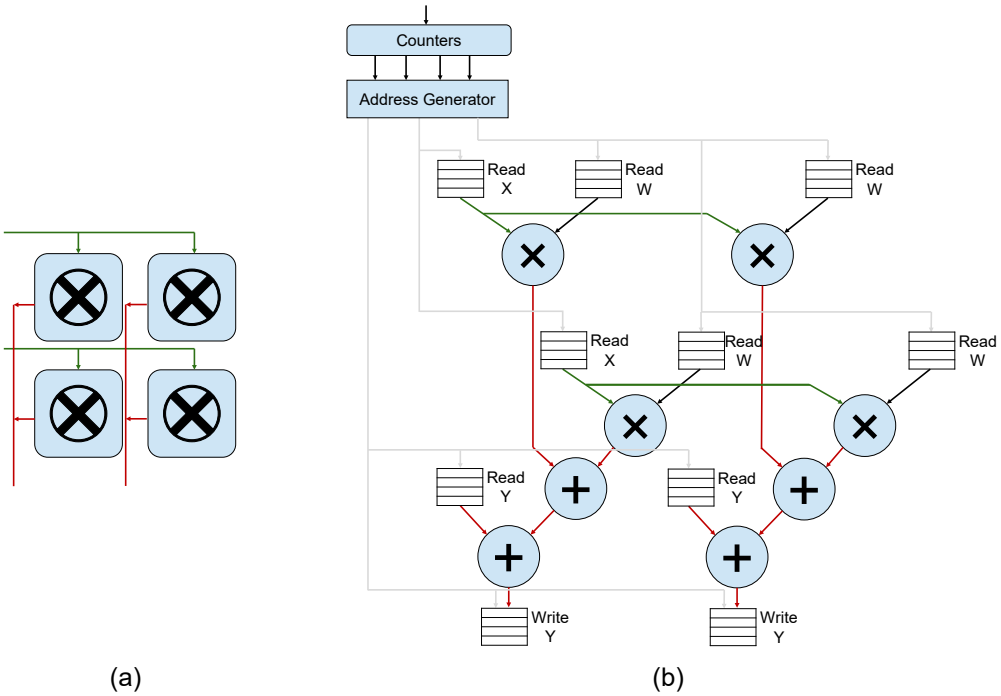

Figure 5-1: An example of an architecture description graph (ADG) (a) and its corresponding detailed architecture graph (DAG) (b). ADG describes the hardware at the FU level and defines the connections between FUs, while DAG opens the black boxes of FUs and describes the hardware using basic primitives like multipliers, adders, and buffers. The boundaries of FUs are also eliminated in DAG. This gives us more flexibility and provides more opportunities for further optimizations.

5.1 Detailed Architecture Graph

The detailed architecture graph (DAG) offers a more detailed view and more fine-grained control over the hardware design. DAG is also a directed acyclic graph in any valid dataflow, but unlike ADG, which adopts FUs as its nodes, DAG selects more low-level hardware primitives as its node. In practice, DAG uses a tree to keep the hierarchy of components for easier hardware floorplan.

Figure 5-2 illustrates the basic primitives of DAG. The primitives include arithmetic logic components like multipliers, adders, logical gates, and comparators, which perform arithmetic operations over two inputs; the data node, which is separated into read and write parts to make sure the graph is acyclic; the multiplexer used to select data from multiple connections; the counter group that generates a series of indexes within the configurable upper and lower bounds, which is used to drive the address

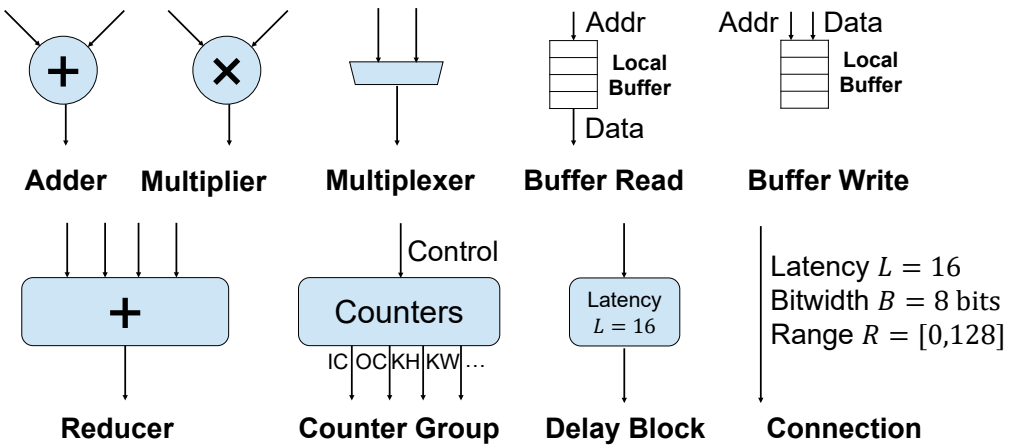

Figure 5-2: Some examples of basic primitives in DAG. The functional units are represented in basic operations like adders and multipliers. Each buffer is split into a read operation and a write operation to prevent cycles in the DAG. The reducer is an intermediate representation of the reduction logic for easier analysis and will be converted to adders later. The counter group provides a series of counters for address generation. The delay block acts like a shift register that delays the input data for a fixed or configurable number of cycles, which is converted from the delayed connection. The connection between components may contain some properties: a non-zero latency L indicates a delayed connection, which is calculated in the delay matching pass; the data range R is used in the width inference pass to derive the bit-width of the signal.

generation logic. We also have data sources representing the control signals decoded from the instructions, and a dummy node, which does not have any logic but a pin to connect multiple wires. The edges between components describe the properties of connections, including bit-width, data range, and data delay (in cycles). In the later passes of LEGO back end, additional components are introduced to DAG: the reduction unit performs arithmetic operations over multiple inputs, which is converted from a chain of associative operations like add, and it will be further transformed to a balanced reduction tree; the delay block delays the input data for a configurable number of cycles, which is converted from the edges with delays.

The translation from ADG to DAG is a rule-based process. The translator reads the computation definition of the FU to first build the arithmetic component, keeps track of the incoming and outgoing connection of the FU, and generates the necessary MUXes or adder in case of multiple connections at the same time. It also reads the

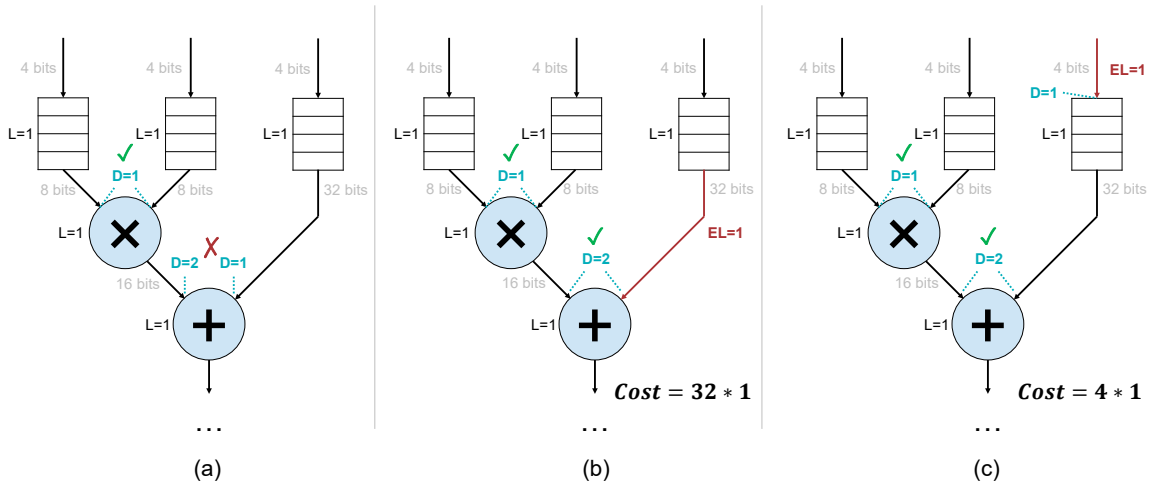

Figure 5-3: Delay matching. The actual implementation of DAG primitives may have some latency L, leading to mismatched delay when two paths to a single component have different latencies (a). This can be solved by inserting extra latency EL into the connections. A greedy algorithm is always inserting the latency to the nearest connection of an unmatched component, which may lead to a higher cost of registers (b). Alternatively, we use a linear program to find the global optimal solution to minimize the register cost (c). In this example, the extra register is moved to the connection with a lower bitwidth on the path, which reduces the cost from 32 to 4.

definition of global computations like address generators, builds the corresponding components, and connects them to the FUs.

5.2 Delay Matching on DAG

One of the major challenges when generating the hardware from the directly translated DAG is that it still represents all computation in combinational logic, which is hard to meet the timing requirement. To relax the timing, an obvious solution is to add pipeline registers to the components according to the complexity. However, simply doing so will lead to incorrect results, since there could be multiple paths from the data source to a given component, and if different delays are introduced on these paths when adding pipeline registers, the component will get mismatched input data and generate incorrect output. Figure 5-3 (a) shows an example of mismatched input.

To address this problem, a delay matching pass is performed to match the delays of multiple paths. Thanks to the acyclic property of DAG, the pass can be in the

topological order. When processing a node v, we could assume all of its preceding nodes u have matched delay D_u due to the topological order, thus we only need to adjust the delay on edge $E_{u,v}$ so that all $D_u + E_{u,v}$ are equal. After this step, we could guarantee node v has matched delay $D_v = D_u + E_{u,v} + L_v$, where L_v is the number of pipeline registers within node v, which is a constant determined by the implementation of hardware primitive. Figure 5-3 (b) shows the results of this greedy algorithm.

We employ the greedy algorithm to find an initial feasible solution to the delay matching problem, and further add consideration to the bit-width of the connections to minimize the number of extra registers added to the edges. From $D_v = D_u + E_{u,v} + L_v$, we could derive that $E_{u,v} = D_v - D_u - L_v$ delay should be added to the edge of (u, v), which is a non-negative number so the constraints are

$$E_{u,v} = D_v - D_u - L_v \geq 0 \tag{5.1}$$

and the optimization target is

$$\min_D \sum_{(u,v)\in E} E_{u,v} * W_{u,v} = \min_D \sum_{(u,v)\in E} (D_v - D_u - L_v) * W_{u,v} \tag{5.2}$$

where $W_{u,v}$ is the bit-width of edge (u, v). This is a linear programming problem that can be solved with any LP solver. Figure 5-3 (c) illustrates the optimal solution find by the linear program. By moving extra latency to low bitwidth connections, the cost can be greatly reduced.

5.3 Broadcast Pin Rewiring on DAG

On DAG, multiple signals can share the same data source, such as the control signals and address signals, and may finally be connected to a reduction logic after several computation logics. To ensure that the delay is consistent, extra latency needs to be added to the path between the data source and the reduction logic, which can

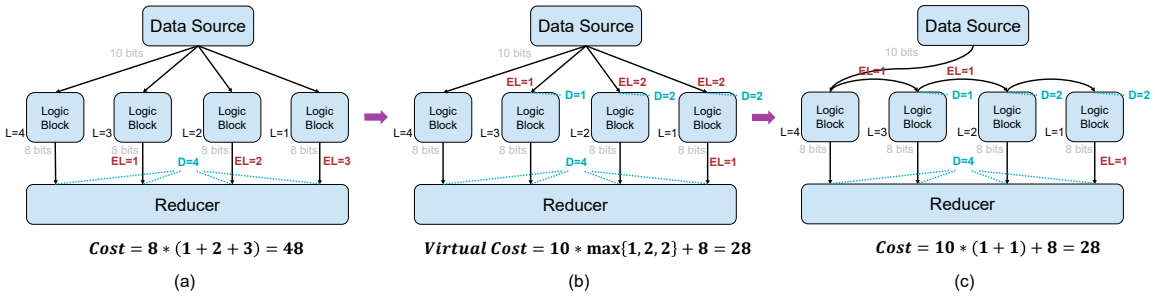

Figure 5-4: Pin rewiring. The pipeline registers (EL denotes the extra latency of them) inserted during delay matching can be suboptimal when there are broadcast pins, and the broadcast may not be directly transformed into a forwarding chain in the case of (a). In (b), we modify the cost function of the linear program to a virtual one that only counts the maximum cost for each broadcast pin. This encourages it to put the pipeline register directly after the broadcast, which allows us to rewire the broadcast connection in (c) using MST.

be optimized by rewiring the broadcast connection into a forward connection. For example, if we insert one cycle latency between A→B and two cycle latency between A→C, we could convert it into a forward chain of A→B→C, in which case only 2 pipeline registers are used instead of 3 in the broadcast case.

However, in practice, the situation can be different as the linear programming solver may not explicitly incorporate the latency on the broadcasted signal. As shown in Figure 5-4 (a), if there is a wire that has a lower bitwidth (8 bits) than the broadcasted signal (10 bits), the solver will prefer it and we might not see any latency on the broadcasted signal. To solve this problem, we propose a three-stage heuristic algorithm for finding good rewiring.

In the first stage, the cost function of the linear program is modified to give preference to the broadcasted signal when adding latency. We use an optimistic estimation of the cost of a broadcasted signal, which is set to the maximum latency between the source and its all destinations. If we could arbitrarily forward data between the destinations, we could always first send the data to a destination with delay D, broadcast it to all destinations with the same delay, and then forward it to another one with delay $D+1$. The overall forward cost will be equal to the maximum latency on the original graph. Figure 5-4 (b) shows the process of cost adjustment.

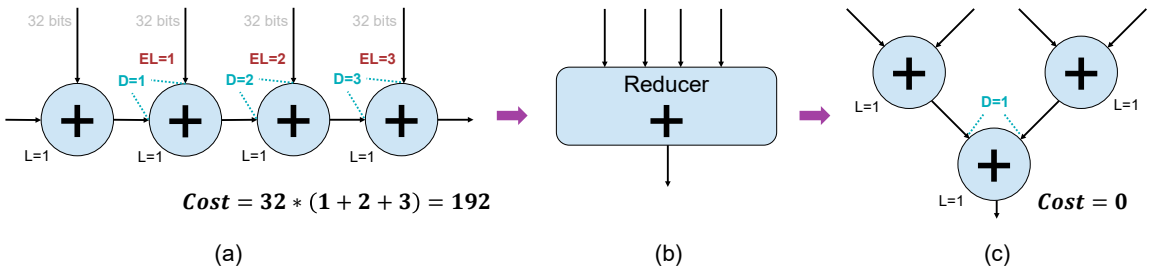

Figure 5-5: Reduction tree extraction. The translation from ADG to DAG usually creates a long chain of adders. To fulfill the requirement of delay matching, extra registers will be inserted into the input connection of each adder, leading to significant overhead (a). We use a reduction tree extraction pass to convert the adder chain to a single reducer component (b). It can be later converted to a more efficient reduction tree (c). The single-component representation in (b) also enables other optimizations such as pin reusing.

The second stage uses a Minimum Spanning Tree (MST) solver for each broadcast source s to perform the rewiring. First, an edge is created from the original source to each destination u to represent the direct connection, of which the cost is the original latency $E_{s,u}$. Then edges are added between spatially adjacent destinations u, v to represent the forwarding, of which the cost is the difference of the delay $|E_{s,u} - E_{s,v}|$. Figure 5-4 (c) shows a possible rewiring solution.

In the third stage, the linear program is re-run on the rewired DAG. This stage is optional but it can ensure a correct and optimal solution after rewiring by redistributing the extra latencies. Usually, a slight improvement in the overall cost can be obtained.

5.4 Reduction Tree Extraction and Pin Reusing

Another transformation pass is the extraction of reduction logic. As shown in Figure 5-5, when translating from ADG, the reduction logic is represented by a long forward chain without any delay, which results in a long chain of computation on DAG and potentially unnecessary high latency. This pass identifies directly connected adders (or any other similar/associative operation like AND/OR) and converts them into a single reduction unit which will later become a balanced tree of reduction.

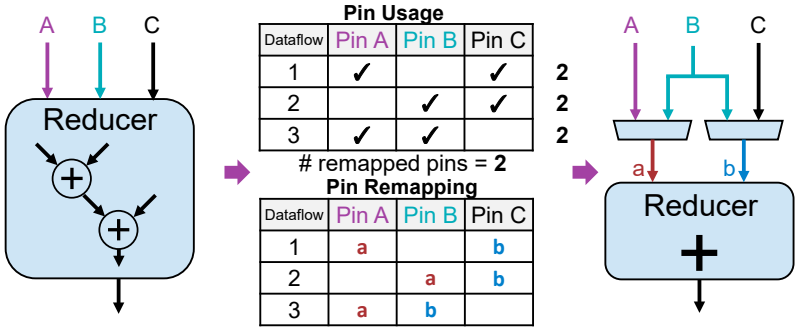

Figure 5-6: Pin reusing can be done after reduction tree extraction. By applying a liveness analysis for each dataflow, we could summarize a table listing the used pin in each case. The number of inputs after remapping will be the maximum number of the used pin in all situations. A pin mapping can be established by a 0-1 integer program, which maps original pins (A, B, C) to the ports (a, b) of the reducer.

The extraction of reduction logic allows us to optimize it further by reusing the pins. In scenarios with multiple dataflow designs, it is possible that not all of the input pins of a reducer are used at the same time. The input pins can be remapped to minimize the cost of the reducer.

The first step is to identify which input pins are used in each dataflow configuration. This is achieved through a liveness analysis. Given a specific dataflow, some input of a MUX component will become a false dependency. By traversing the DAG, we can determine whether a pin contributes to any true dependency in the final output (output buffer). The information is recorded in a pin usage table for deriving the number of remapped pins, as shown in Figure 5-6.

The next step is to build a pin remapping. This is done by a 0-1 integer program. We use 0-1 variables $C(i, j, k)$ to control if original pin i should be mapped to physical pin j in dataflow k, and set $A(k)$ to represent the active pins in dataflow k. We need to make sure each active pin must connect to one physical pin: $\sum_j C(i, j, k) = 1$ (if $i \in A(k)$), and each physical pin only connects to at most one input: $\sum_i C(i, j, k) \leq 1$. The optimization objective is to minimize the connections $\sum_{i,j,k} C(i, j, k)$.

The DAG will then be built based on the pin mapping. If there are multiple input maps to one pin of the reducer, a MUX will be used. Since a MUX is much more lightweight than an adder on ASIC, the pin reusing could effectively reduce the area

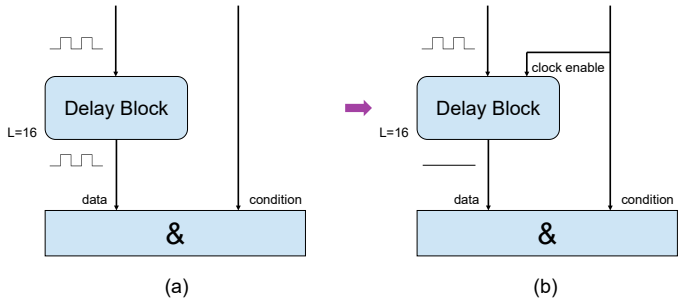

Figure 5-7: Power gating. When generating hardware with multiple spatial dataflows, the connection between FUs might not always be used, leading to the redundant power consumption of registers. The power gating pass identifies the potentially unused connection and gates the clock signal of registers to improve energy efficiency.

and power.

5.5 Other Transformation and Elaboration Passes on DAG

The design of DAG enables many useful elaboration and optimization passes.

For example, as shown in Figure 5-7, a power-gating pass can reduce the power consumption of unused connections between FUs, which is a side effect of dataflow switching. The power gating optimizer reuses the liveness analyzer in Section 5.4 to find out unused pins. It adds clock-enable signals in front of each delay block, which eliminates the unused data toggle.

Another example elaboration pass is automatic bit-width inference. When building the DAG, we did not specify the essential bit-width information on edges. This pass can be used to automatically derive them from the data sources. We first define a width inference function for each component, which could generate the output data range from the input range (min, max), the width inference pass will go through all components in topological order to annotate the data range to each edge, which can be further converted to bit-width information.

We also add other elaboration passes in the later stage of LEGO back end, such as delay block transformation, which converts the delay on edges to delay blocks, and

cross hierarchy connection, which create extra pins to help connect components in different hierarchy level.

LEGO back end can also be extended to better support specific hardware platforms like FPGA. A pattern-matching-based transformation pass can be used to better utilize the hardware resources, like MAC fusion pass, which fuse the adder to the adjacent multiplier, forming a MAC component, and DSP fusion pass, which could fuse two 8-bit multipliers into a larger DSP component.

Chapter 6

Evaluation

6.1 Evaluation Methodology

We implemented the LEGO framework in C++ with HiGHS [12] as the linear programming solver in the back end. We built all of the DAG primitives and NoC components in SpinalHDL and use it to generate the synthesizable Verilog code from the optimized DAG. We synthesize the generated hardware using Synopsys Design Compiler with TSMC 28nm library and use CACTI [21] to model SRAM. Figure 6-1 shows the breakdown of hardware. For latency estimation, we built a performance simulator for the FU array and NoC in the front end to fastly predict the cycle numbers of computation and memory access, which are verified with the RTL simulation.

We evaluated LEGO with multiple popular neural network models including AlexNet [14], MobileNetV2 [27], Resnet50 [10], EfficientNetV2 [30], BERT [6], GPT-2 [25], CoAtNet [4], and modern generative AI models including DDPM [11], Stable Diffusion [26], and LLaMA-7B [32]. We use a sentence length of 16 when evaluating BERT. For GPT-2 and LLaMA-7B, we set the input length to 1000 and measure the latency of generating one token. Large language models can be inferenced in batches on cloud, so we set the batch size to 1 and 32 for LLaMA-7B. We also evaluated LEGO with different tensor kernels, including 2D Conv, GEMM, and MTTKRP on multiple dataflow settings.

The state-of-the-art open-source NN accelerator generator, Gemmini [7], is adopted

Table 6.1: Performance and energy efficiency of large generative models running on LEGO-ICOC-1K with 32*32=1024 FUs, 576 KB buffer, 32 post-processing units, and 32GB/s memory bandwidth. The on-chip area and power are 3.95 mm^2 and 601 mW. LLaMA-7B is bounded by memory bandwidth.

Model	**DDPM**	**Stable Diffusion**	**LLaMA-7B** bs=1	**LLaMA-7B** bs=32
Utilization	92.9%	80.2%	3.1%	42.9%
Perf. (GOP/s)	1903	1642	63	878
Energy Eff. (GOPS/W)	3165	2731	105	1461

as our baseline. For a fair comparison, we configured LEGO to use a similar hardware resource to Gemmini. Both have 256 FUs, 256 KB of buffer, and a 128-bit memory bus with 16GB/s bandwidth.

6.2 Experimental Results

Figure 6-2 and Figure 6-3 compare the performance and energy efficiency running neural network models. LEGO achieves nearly theoretical maximum performance on Resnet50, EfficientNetV2, BERT, and CoAtNet models. Both Gemmini and LEGO do not perform well on GPT-2 since generative models like GPT-2 are typically bounded by memory bandwidth. On average, LEGO achieves 3.2× speedup and 2.4× energy saving over Gemmini. The speedup mainly comes from 2 parts: (i) LEGO front end integrates a fast and accurate performance simulator, which could guide the scheduler to find the optimal mapping policy to the hardware; (ii) the support of flexible connection enables dynamic dataflow change, which extends the search space for the scheduler. The speedup is significant on depthwise convolutional layers since LEGO could switch to OH-OW dataflow on these layers.

For modern generative AI models, we use a larger and simpler architecture generated by LEGO, which has 1024 FUs. As shown in Table 6.1, LEGO achieves >90% utilization on DDPM and >80% on Stable Diffusion. LLaMA-7B model has very low operational intensity, especially in attention layers, of which the performance is

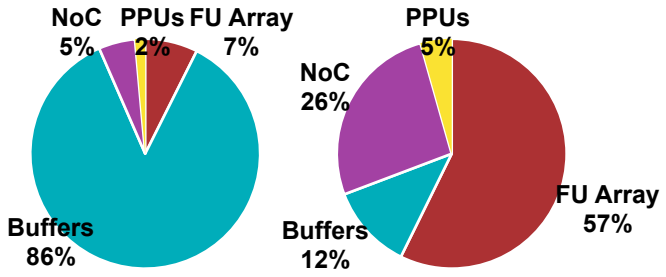

Figure 6-1: Area (left, 1.76mm^2 in total) and on-chip power (right, 285mW in total) breakdown of LEGO-MNICOC. The hardware is configured to match the resource of Gemmini.

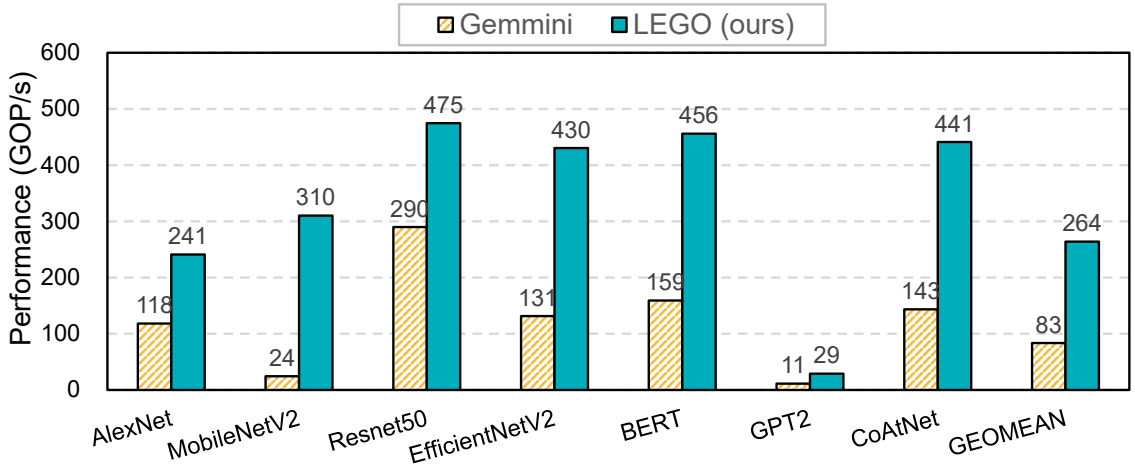

Figure 6-2: Performance comparison between Gemmini and LEGO. The performance numbers of LEGO are end-to-end results and include the post-processing units (PPUs), which account for 0.5% to 7.2% of the latency. LEGO achieved an average of 3.2× speedup over Gemmini. Both Gemmini and LEGO are bounded by memory bandwidth on GPT2. LEGO performs much better on MobileNetV2 due to its efficient support of depthwise convolution by dataflow switching.

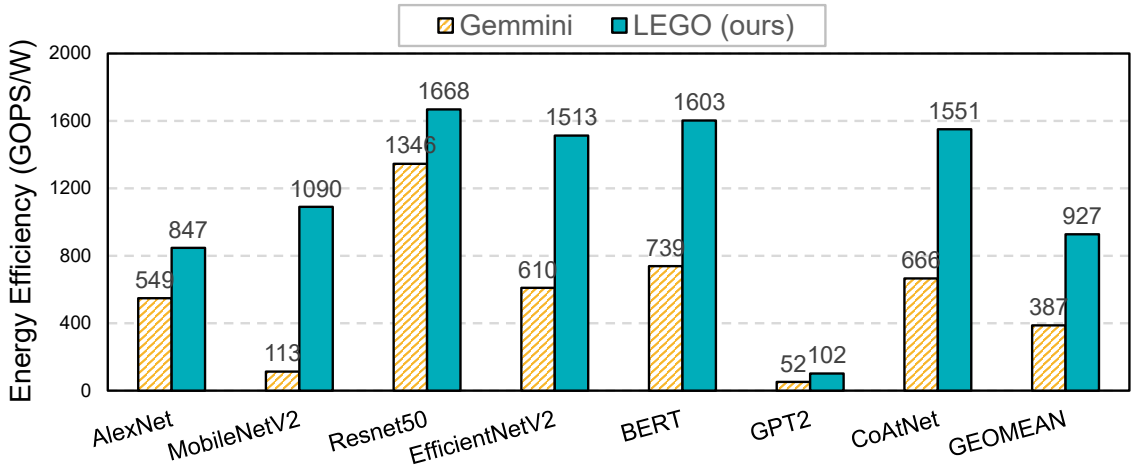

Figure 6-3: On-chip energy efficiency comparison between Gemmini and LEGO. LEGO achieved an average of 2.4× energy savings.

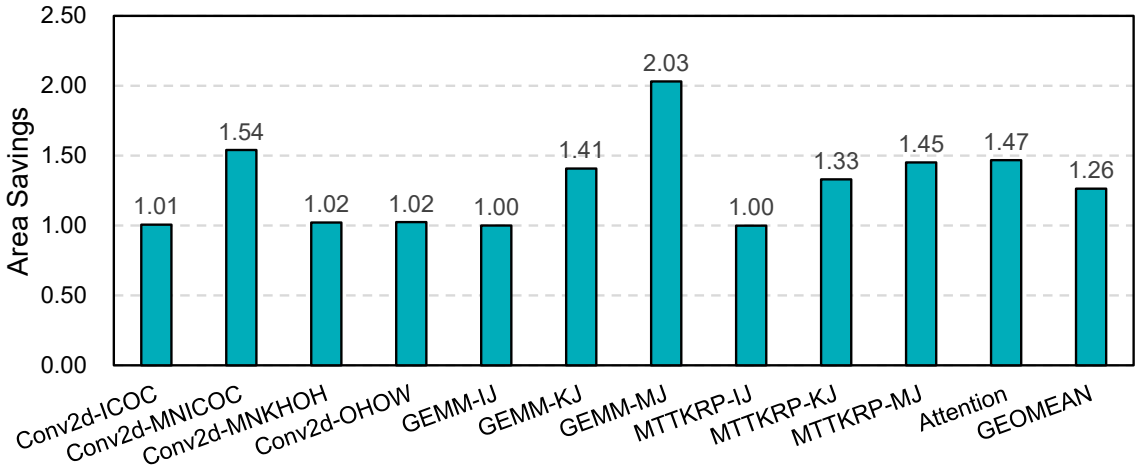

Figure 6-4: The area savings of LEGO optimizations on tensor kernels named as *Operation-Dataflow*. M and N represent dynamically switchable dataflow. This compares the baseline (delay matching only, which is mandatory for the timing requirement) with the optimized architecture (with pin reusing, reduction tree optimizations, and power gating). The improvement is significant for dynamic dataflows. The average saving is 1.26× and can be up to 2.03×.

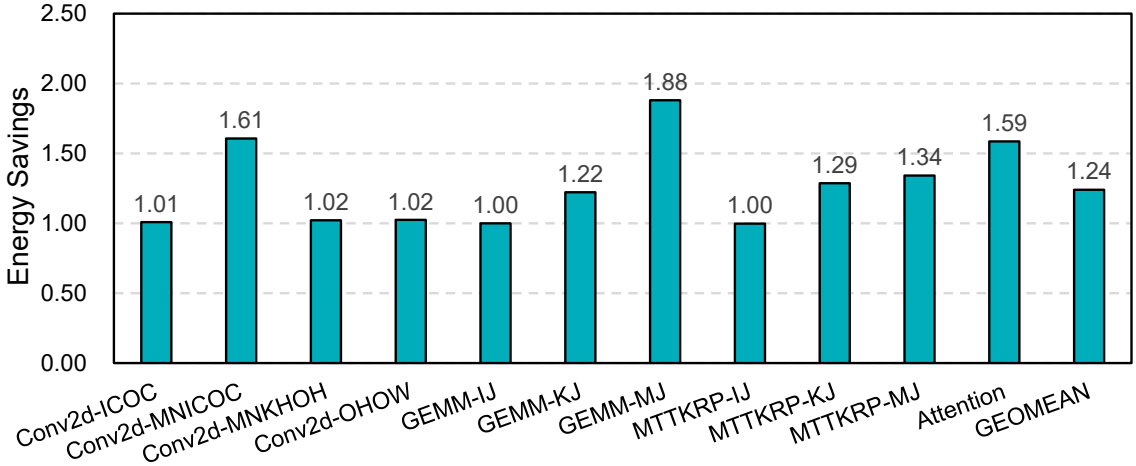

Figure 6-5: The energy savings of LEGO optimizations on tensor kernels. The average saving is 1.24× and can be up to 1.88×.

heavily bounded by the DRAM bandwidth.

Figure 6-4 and Figure 6-5 compare the area and energy savings before and after the optimization passes in LEGO on different tensor kernels. On average, LEGO achieves 1.29× area savings and 1.26× energy savings. The improvement is especially remarkable for tensor kernels with multiple dataflows since connection topology in these architectures is more complex and provides more opportunities for optimization.

We evaluated the quality of the generated hardware by comparing LEGO against human-expert-designed accelerators, including Eyeriss [1] and NVDLA [41], using the same settings. The settings and synthesis results are listed in Table 6.2. When scaled to the same technology node, the automatically generated implementation achieves comparable area and power to human design. Additionally, compared to Eyeriss, LEGO-generated architecture achieved lower power consumption due to the reduction of power in scratchpads. This is because the data read from buffers are shared between FUs by the inter-FU connections in LEGO.

We also evaluated the running time performance of LEGO by measuring the time spent by different stages of LEGO on our Intel i9-9900K desktop CPU. During the generation of LEGO-MNICOC architecture, LEGO spent 4 seconds in frontend, 70 seconds in backend, and 68 seconds in SpinalHDL and Verilator compilation. This result is comparable to Gemmini (235 seconds) and is negligible compared to the synthesis tool ($>$ 1 hour).

To better understand the contribution of the optimization techniques in LEGO, we perform a breakdown analysis on a typical tensor kernel Attention. Figure 6-6 shows the area and energy efficiency breakdown of each step. Both reduction tree extraction and broadcast can reduce the number of pipeline registers, resulting in 5% and 6% area saving. The power gating technique significantly reduces the toggle rate on unused paths, leading to 19% of power saving. The pin reusing applied to the reduction tree significantly saves the cost of adders and the extra pipeline register for them, achieving 22% area saving and 29% power saving.

Table 6.2: Comparison between human-designed and LEGO-generated architectures. We configure LEGO to use the identical dataflow as the baseline so the theoretical performance is the same. The power numbers of LEGO are peak power, which means the FU array is under nearly 100% utilization. The area and power of LEGO are comparable to human-designed ones. The power of LEGO-KHOH is even lower than Eyeriss due to the reuse of data read from buffers.

Architecture	**Eyeriss**	**LEGO-KHOH**	**NVDLA**	**LEGO-ICOC**
Dataflow	KH-OH Parallel		IC-OC Parallel	
#FUs	168		256	
Frequency	200 MHz		1 GHz	
Technology	60nm	65nm	16nm	16nm*
Area (mm^2)	9.6	7.4	1.0	0.9
Power (mW)	278	112	135	94

* projected from 28nm [38]

6.3 Extensibility Study: Low-bitwidth Quantization Support for LLM

Low-bitwidth quantization is a trending topic to reduce the size of neural network models. In this work, we choose VS-Quant [3], Microexponent (MX) [5], and an FP4-based groupwise quantization (FP4-INT4) to demonstrate the extensibility of LEGO when dealing with these emerging techniques.

The flexibility of LEGO allows us to convert an existing INT8 architecture into FP4 architecture without much effort, thanks to the user-defined FU support in LEGO. For example, in FP4-INT4, a basic functional unit performs the dot product between a group of activations and weights, and it can be described using the following pseudo-code:

```
output := scaleAct * scaleWgt * (
          (act, wgt).zipped.map(_ * _).reduce(_ + _) +
          zeropointAct * sumWgt)
```

Unlike human-designed architectures, where the engineer needs to specify the bitwidth of each signal, carefully design the pipeline, and use a balanced tree or

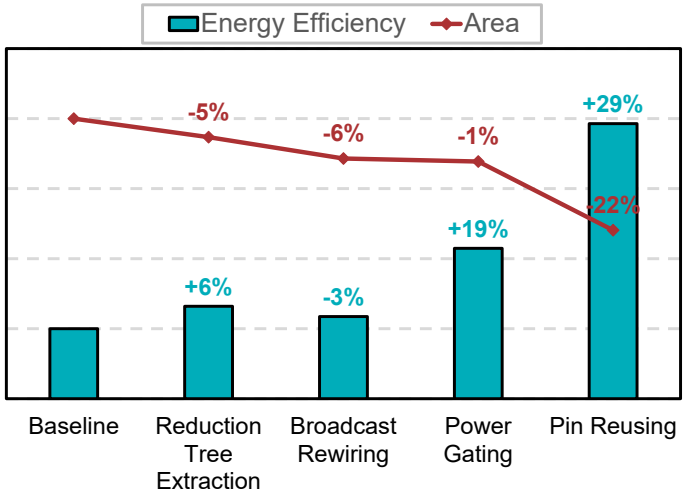

Figure 6-6: Performance breakdown of LEGO optimizations on attention tensor kernel. Reduction tree extraction and broadcast pin rewiring optimize area by reducing the number of pipeline registers. Power gating optimizes energy by gating the unused connection. Pin reusing optimizes both by reducing the number of adders.

systolic array to expand the adder logic, LEGO backend could systemically analyze the hardware and perform these optimizations automatically, which greatly reduce the requirement of human effort.

Table 6.3 shows the comparison results of the FU array of multiple low-bitwidth quantization architectures on LLaMA-7B(with batch size = 1 and 32 and memory bandwidth = 32 GB/s). Compared to the INT8 baseline, 4-bit quantization could reduce the complexity of multipliers, resulting in up to 2.2x area saving and 2.5x power saving. 4-bit quantization also reduce the off-chip memory traffic, which is the main bottleneck during the inference of LLMs, resulting in 1.4-2.0x speedup and overall 3.5x-5.1x energy efficiency improvement.

Table 6.3: Comparison of FU array of INT8 and multiple low-bitwidth architectures generated by LEGO. The performance is evaluated on LLaMA-7B model. The latency results are end-to-end and mainly bounded by the memory traffic of dense layers. (The proportion of activation functions ranges from 0.06% to 0.2% when bs=1 and 0.8% to 2.3% when bs=32.) Low-bitwidth architecture could save area/power by reducing computation complexity and improve performance by reducing memory access. We also compared the accuracy (perplexity on Wikitext2, lower is better) using the weights of Llama2. FP4-INT4 architecture has a much higher accuracy than VSQ-INT4 and MX4 with a comparable hardware cost.

Architecture	**FP16**	**INT8**	**FP4-INT4**	**VSQ-INT4**	**MX4**	**MX6**	**MX9**
FU Array Size	128*8						
Group Size	-	-	64	16	16	16	16
Area (mm^2)	0.629	0.241	0.114	0.109	0.108	0.173	0.297
Power (mW)	674	425	167	179	167	262	487
Perf. bs=1 (GOP/s)	32	63	114	118	125	84	56
Eff. bs=1 (GOP/s/W)	47	148	682	664	752	319	115
Perf. bs=32 (GOP/s)	429	851	1194	1213	1239	1022	757
Eff. bs=32 (GOP/s/W)	637	2002	7138	6796	7435	3899	1554
Perplexity (Lower is better)	5.47	5.47	5.93	11.10	35.85	5.55	5.48

Chapter 7

Conclusion

We provide an automatic design methodology, LEGO for highly flexible spatial accelerators without templates. In summary, the contributions of this work are as follows:

- Provides an end-to-end solution, LEGO framework, that streamlines the hardware development process from tensor applications to RTL code based on hierarchical LEGO spatial architecture.
- Introduces a high-level hardware representation based on the mappings between the iteration domain and (par)for-loop instances.
- Builds an FU-level spatial architecture graph by solving integer linear equations and optimizes the graph via an MST-based algorithm and a BFS-based heuristic algorithm to minimize the data path overhead.
- Optimizes the primitive-level architecture graph via a set of linear-programming-based algorithms to reduce register cost and unused connection overhead.
- LEGO outperforms Gemmini [7] with 3.2× speedup and 2.4× energy efficiency. LEGO can generate one architecture for diverse generative AI applications including the LLaMA foundation model and stable diffusion model.

In other words, LEGO enables a larger hardware design space, and more efficient development of spatial accelerators, allowing for greater potential for customization for diverse tensor applications.

Bibliography

[1] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[2] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3075–3084, 2019.

[3] Steve Dai, Rangha Venkatesan, Mark Ren, Brian Zimmer, William Dally, and Brucek Khailany. Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference. *Proceedings of Machine Learning and Systems*, 3:873–884, 2021.

[4] Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *Advances in Neural Information Processing Systems*, 34:3965–3977, 2021.

[5] Bita Darvish Rouhani, Ritchie Zhao, Venmugil Elango, Rasoul Shafipour, Mathew Hall, Maral Mesmakhosroshahi, Ankit More, Levi Melnick, Maximilian Golub, Girish Varatkar, et al. With shared microexponents, a little shifting goes a long way. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774. IEEE, 2021.

[8] GitHub. GitHub Copilot: AI pair programmer. `https://copilot.github.com/`, 2021.

[9] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. A$\hat{3}$: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

[12] Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[13] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[15] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.

[16] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

[17] Yujun Lin, Mengtian Yang, and Song Han. Naas: Neural accelerator architecture search. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1051–1056. IEEE, 2021.

[18] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. Pointacc: Efficient point cloud accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–461, 2021.

[19] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. Tenet: A framework for modeling tensor

dataflow based on relation-centric notation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 720–733. IEEE, 2021.

[20] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 977–991, 2021.

[21] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[22] OpenAI. ChatGPT: a large language model trained on the GPT-3.5 architecture. `https://openai.com/`, 2021.

[23] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[24] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[25] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[26] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[27] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[28] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*, 2016.

[29] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

[30] Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *International conference on machine learning*, pages 10096–10106. PMLR, 2021.

[31] Robert Endre Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.

[32] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[34] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[35] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[36] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 93–104, 2021.

[37] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[38] Shien-Yang Wu, Colin Yu Lin, MC Chiang, JJ Liaw, JY Cheng, SH Yang, Ming Liang, Tadakazu Miyashita, CH Tsai, BC Hsu, et al. A 16nm finfet cmos technology for mobile soc and computing applications. In *2013 IEEE International Electron Devices Meeting*, pages 9–1. IEEE, 2013.

[39] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, Vikas Chandra, Weiwen Jiang, and Yiyu Shi. Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[40] Yongan Zhang, Yonggan Fu, Weiwen Jiang, Chaojian Li, Haoran You, Meng Li, Vikas Chandra, and Yingyan Lin. Dna: Differentiable network-accelerator co-search. *arXiv preprint arXiv:2010.14778*, 2020.

[41] Gaofeng Zhou, Jianyang Zhou, and Haijun Lin. Research on nvidia deep learning accelerator. In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 192–195. IEEE, 2018.