POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A Framework for Exploring Fused Layers in DNN Dataflows for Spatial Accelerators

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Matteo Salvatore Currò**

Student ID: 223648
Advisor: Prof. Cristina Silvano
Co-advisors: Prof. Leandro Fiorin
Academic Year: 2024-25

Abstract

Deep neural networks (DNNs) demand immense computational resources, with off-chip DRAM accesses constituting the primary energy bottleneck during inference. While standard layer-by-layer execution requires redundant DRAM transfers for intermediate activations, layer fusion mitigates this by keeping intermediate data on-chip through interleaved scheduling. However, optimizing fused-layer execution introduces cascading inter-layer dependencies and drastically inflates the mapping search space, making the systematic exploration exceptionally challenging.

This thesis formally characterizes the complex design space of multi-layer fused DNN execution on spatial accelerators. To systematically navigate this space, an analytical modeling framework is developed. By introducing dimension aliasing and decoupling, the methodology unifies multi-layer computations into a single loop nest while strictly preserving independent, per-layer mapping freedom. Furthermore, an exact producer-consumer constraint-based pruning strategy rigorously eliminates infeasible mappings. The analytical cost model is extended to compute per-layer memory operations, latency, and energy, and is structurally validated against the DepFiN hardware accelerator.

Extensive exploration using DepFiN-like and Eyeriss-like architectures was conducted across activation-dominant and weight-dominant workloads. The findings reveal that while full fusion successfully reduces total DRAM traffic by 25% to 95% and significantly lowers latency for activation-dominant workloads (up to 42–48% of savings), it consistently incurs a severe energy penalty for weight-dominant networks due to exponentially enlarged on-chip memory requirements. Conversely, partial pairwise fusion emerges as a highly practical compromise, capturing nearly all latency benefits while substantially mitigating the on-chip memory overhead. Ultimately, this work demonstrates that layer fusion is primarily a workload-dependent latency optimization rather than a universal energy solution, providing designers with a scalable, formalized methodology to systematically navigate these complex architectural trade-offs.

Keywords: deep neural networks, spatial architectures, layer fusion, design-space exploration

Abstract in lingua italiana

Le reti neurali profonde (DNN) richiedono immense risorse computazionali, con gli accessi alla DRAM *off-chip* che costituiscono il principale collo di bottiglia energetico durante l'inferenza. Mentre l'esecuzione standard livello per livello *(layer-by-layer*) richiede trasferimenti DRAM ridondanti per le attivazioni intermedie, la fusione dei *layer* (*layer* fusion) mitiga questo problema mantenendo i dati intermedi on-chip attraverso uno scheduling interleaved. Tuttavia, l'ottimizzazione dell'esecuzione a *layer* fusi introduce dipendenze a cascata tra i layer e ingrandisce drasticamente lo spazio di ricerca del mapping, rendendone l'esplorazione sistematica eccezionalmente complessa.

Questa tesi caratterizza formalmente il complesso spazio di progetto dell'esecuzione di DNN a più layer fusi su acceleratori spaziali. Per navigare sistematicamente in questo spazio, viene sviluppato un *framework* di modellazione analitica. Attraverso l'introduzione dell'aliasing e del disaccoppiamento delle dimensioni, la metodologia unifica i calcoli multi-*layer* in un singolo *loop nest*, preservando rigorosamente la libertà di mapping indipendente per ogni layer. Inoltre, un'esatta strategia di pruning basata sui vincoli produttore-consumatore elimina rigorosamente i *mapping* irrealizzabili. Il modello analitico di costo è stato esteso per calcolare le operazioni di memoria, la latenza e l'energia per singolo layer, ed è validato strutturalmente rispetto all'acceleratore *hardware* DepFiN.

È stata condotta un'ampia esplorazione utilizzando architetture simili a DepFiN ed Eyeriss su *workload* dominati dalle attivazioni e dominati dai pesi. I risultati rivelano che, sebbene la fusione completa riduca con successo il traffico DRAM totale dal 25% al 95% e diminuisca significativamente la latenza per i workload dominati dalle attivazioni (con risparmi fino al 42-48%), essa comporta costantemente una grave penalizzazione energetica per le reti dominate dai pesi, a causa dell'aumento esponenziale dei requisiti di memoria on-chip. Al contrario, la fusione parziale a coppie emerge come un compromesso altamente pratico, catturando quasi tutti i benefici in termini di latenza e mitigando sostanzialmente l'overhead di memoria on-chip. In definitiva, questo lavoro dimostra che la fusione dei layer è principalmente un'ottimizzazione della latenza dipendente dal workload, piuttosto che una soluzione energetica universale, fornendo ai progettisti una metodologia formalizzata e scalabile per navigare sistematicamente questi complessi trade-off architetturali.

Parole chiave: Deep Neural Network, Spatial Accelerators, Fusione di Layer, esplorazione del design space

Contents

1 | Introduction

1.1. Motivations

Deep neural networks (DNNs) have become the dominant computational paradigm across a wide range of applications, from image classification and object detection to natural language processing and autonomous driving [16]. The computational cost of modern DNNs, however, is immense: a single inference pass of a large convolutional network such as VGG16 requires billions of multiply-and-accumulate (MAC) operations, while training demands orders of magnitude more [4]. While general-purpose processors and graphics processing units (GPUs) deliver high computational throughput, their rigid memory hierarchies often struggle to meet the stringent energy-efficiency (performance-per-watt) requirements of specialized edge devices. This bottleneck has motivated the development of dedicated *spatial architectures* (SAs): custom accelerators that exploit massive data-level parallelism through arrays of processing elements (PEs) interconnected by customized on-chip memory networks.

The classic execution model for a neural network on a SA is the layer-by-layer execution model, where the neural network is computed strictly one layer at a time. This approach introduces a severe constraint: the output activation tensor must be entirely written to off-chip DRAM after one layer completes, only to be read back when the next layer begins. This creates a "memory wall": off-chip DRAM accesses are roughly two orders of magnitude more expensive than on-chip SRAM accesses in terms of energy. This traffic pattern typically dominates the total energy budget for activation-heavy workloads. Furthermore, the limited bandwidth of DRAM frequently leads to processing stalls when handling heavy read and write traffic of intermediate data, severely degrading latency.

Layer fusion has emerged as a promising strategy to address this bottleneck by computing multiple consecutive layers in a single, interleaved schedule. By tiling the execution, intermediate activations can be produced and consumed entirely within on-chip buffers in localized chunks, effectively eliminating the round trip through external DRAM. Hardware implementations such as DepFiN [10] have demonstrated that depth-first fused execution

can achieve significant energy savings and latency reductions for certain workloads.

However, a crucial challenge remains even for single-layer execution: extracting optimal performance and energy efficiency from these hardware designs relies critically on the *mapping*, a schedule that determines the order in which loop iterations are executed, how data is tiled and distributed across PEs, and what data is retained at each level of the memory hierarchy. Even for a single convolutional layer on a moderately complex SA, the set of all possible mappings (the map space) exceeds 10^{19} candidates. This staggering complexity has driven the development of fast analytical cost models and automated mappers to efficiently search for optimal single-layer schedules. Furthermore, fusing L layers exponentially multiplies the number of loop dimensions, introduces strict producer–consumer dependencies, and creates new opportunities for data reuse. This dimensional proliferation causes an astronomical growth in the map space. For instance, an 10-layer fusion of FSRCNN on the DepFiN architecture expands the map space to over 10^{515} candidates, which is more than 400 orders of magnitude larger than its single-layer equivalent [22]. To tame this intractable space, it is fundamental to study the cascading tile-size dependencies between producer and consumer layers. Enforcing these dependencies can rigorously prune the map space of physically infeasible mappings that an automated mapper would otherwise needlessly evaluate during the search for optimality. Moreover, maximizing data reuse requires optimizing not only the intra-layer (within a single layer) schedules but also the inter-layer tile mappings, necessitating a systematic analysis of the different types of buffered and immediate reuse mechanisms that fused execution unlocks.

Despite the increase in complexity, since the mapping of a single layer is natively expressed as a loop nest, it is highly desirable to maintain this representational consistency across the extended map space. Describing the fused-layer mapping as a single, unified loop nest via the Extended Einsum notation [8] provides an exact, coherent mathematical representation. While existing fused-layer exploration frameworks, such as DeFiNES [20] and LoopTree [9], have made significant progress in exploring the fused dataflow design space, no prior tool represents both single-layer and fused-layer mappings under a unified, coherent loop-nest abstraction.

Therefore, the core motivation of this thesis is to formally explore the fused-layer dataflow design space through a unified analytical cost model. This work seeks to rigorously study the architectural pros and cons of implementing fused layers, analyzing how different hardware memory hierarchies and workload characteristics (such as activation versus weight dominance) dictate its success or failure.

1.2. Objectives

The primary objective of this study is to formally explore the design space of multi-layer layer fusion and quantify its inherent performance and energy trade-offs. To navigate this space, our thesis extends the FactorFlow analytical mapping framework for single layer. The specific objectives are to:

1. **Formalize the Fused-Layer Design Space:** Characterize the theoretical complexities of layer fusion, including dimension proliferation, operand residency rules, and the cascading inter-layer tile dependencies that dictate the reuse-recompute trade-off.
2. **Develop Constraint-Based Pruning:** Tame the astronomical map-space explosion by establishing an exact, scheduling pruning strategy that enforces strict producer–consumer tile-size inequalities at every level of the hardware hierarchy.
3. **Construct an Analytical Evaluation Engine:** Extend FactorFlow's existing factor-based abstraction and extended Einsum notation to unify multi-layer loop nests (via aliasing and decoupling) without sacrificing intra-layer mapping independence. Generalize the single-layer cost model to assess memory operations, energy, and latency on a per-layer basis.
4. **Validate the Model:** Validate the extended analytical framework against a published hardware baseline (DepFiN), demonstrating that the predictions for complex, deep fusion topologies match the reference.
5. **Quantify the Pros and Cons of Fusion:** Conduct a comprehensive design-space exploration across different spatial architectures (DepFiN-Like and Eyeriss-Like) and CNN workloads (activation-dominant vs. weight-dominant). By comparing full fusion, partial fusion, and single-layer execution modes, this objective aims to definitively isolate the architectural and algorithmic conditions under which layer fusion provides a tangible benefit versus a detrimental overhead.

1.3. Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 – Background introduces the foundational concepts: artificial intelligence and deep learning, DNN workloads and the extended Einsum notation, hardware acceleration with spatial architectures, the mapping problem and map-space explo-

sion, analytical cost modeling, and the mechanics of layer fusion.

Chapter 3 – Fused-Layer Mapping Design Space formalizes the core theoretical challenges of the fused-layer mapping problem: the exponential proliferation of dimensions, cascading tile dependencies, operand categorization, memory bypass rules, and the fundamental reuse-recompute trade-offs that dictate map-space feasibility.

Chapter 4 – Related Works examines the state of the art in layer-by-layer dataflow exploration tools (Timeloop, FactorFlow) and fused-layer evaluation frameworks (DeFiNES, LoopTree), contextualizing them within the broader landscape of layer fusion methodologies.

Chapter 5 – Proposed Solution presents the analytical framework developed to explore the design space: the aliasing and decoupling abstraction, the exact constraint-based pruning, the generalized cost model, and software implementation details.

Chapter 6 – Validation validates the extended analytical model against the DepFiN accelerator, analyzing the agreement between the model's predicted data-movement and energy metrics and the published figures from a baseline fused workload experiment.

Chapter 7 – Experimental Results presents the comprehensive design-space exploration. It details the setup, single-architecture sensitivity analyses and evaluates the critical pros and cons of full, partial, and single-layer execution across varied architectural constraints.

Chapter 8 – Conclusions and Future Work summarizes the theoretical and empirical findings of the study and outlines directions for future research toward automated heuristic mappers.

2 | Background

This chapter introduces the foundational concepts underpinning this thesis. Section 2.1 provides a brief overview of artificial intelligence, neural networks, and deep learning. Section 2.2 formalizes DNN workloads and introduces the extended Einsum notation used throughout this work. Section 2.3 describes hardware acceleration for deep learning and the abstraction of spatial architectures. Section 2.4 defines the mapping problem and the degrees of freedom available to a mapper. Section 2.5 analyzes the combinatorial explosion of the resulting map-space. Section 2.6 discusses cost modeling for spatial architectures. Finally, Section 2.7 introduces the concept of layer fusion and its implications for memory traffic reduction.

2.1. Artificial Intelligence

Throughout human history the aspiration to replicate human thought artificially has been a persistent theme; testimony of this desire dates II century B.C.E. The myth of Talos, a guardian constructed entirely of bronze engineered solely for defense and in its non-programmed desire for immortality [3] furnishes the mythological foundation for the Greek concept of the *automaton*, an entity that moves of its own accord, and has been invoked to illustrate the ability to solve problems autonomously, not through the formal specification of rules, but by learning from experience which choice is the most appropriate.

This inductive process, which enables the extraction of patterns from data, is known as *machine learning*. As an inductive method, it is heavily dependent on the quantity and quality of the data from which it learns, and in particular on the *representational form* of that data. Initially, the selection of representations, called *features*, had to be performed manually by domain experts; this limitation was subsequently overcome by *deep learning*, in which complex concepts are represented hierarchically in terms of simpler ones. When visualized, this hierarchy of concepts forms a deep computational graph, leading to the term **Deep Learning** (DL). The fundamental principle of DL is to learn an efficient *latent representation* of the input data, one designed to capture the essential semantic information required for a given task and to transform the problem into a space where a

solution can be more easily discovered.

2.1.1. From Perceptrons to Deep Neural Networks

The cornerstone of modern neural networks is the ***perceptron***, the first tangible implementation of an artificial learning neuron. The concept was developed throughout the 1950s and 1960s by Frank Rosenblatt [25], inspired by earlier work from Warren McCulloch and Walter Pitts.

A perceptron takes several binary inputs, $x_1, x_2, \ldots$, and produces a single binary output. Rosenblatt proposed a simple rule to compute this output by introducing *weights*, $w_1, w_2, \ldots$, which are real numbers expressing the relative importance of each input. The neuron's output, either 0 or 1, is determined by whether the weighted sum of its inputs, $\sum_j w_j x_j$, falls below or exceeds some *threshold value*. Like the weights, the threshold is a real-valued parameter of the neuron. This decision rule can be expressed algebraically as:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold}, \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold}. \end{cases} \tag{2.1}$$

To simplify the notation, two standard substitutions are made. First, the summation is rewritten as a dot product, $\mathbf{w} \cdot \mathbf{x} \equiv \sum_j w_j x_j$, where $\mathbf{w}$ and $\mathbf{x}$ are vectors of the weights and inputs, respectively. Second, the threshold is moved to the other side of the inequality and replaced by the neuron's *bias*, defined as $b \equiv -\text{threshold}$. The bias can be thought of as a measure of how easily the perceptron can be made to output a 1, or, in biological terms, to "fire." Using the bias, the perceptron rule becomes:

$$\text{output} = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0, \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0. \end{cases} \tag{2.2}$$

This mathematical model can be viewed as a simple decision-making unit that evaluates and combines multiple pieces of evidence through weighted contributions. By tuning its weights and bias, the perceptron can be adapted to represent various types of decision processes. Its ability to *learn* from data by iteratively adjusting these parameters constitutes the key advancement over the McCulloch–Pitts (MP) neuron model, which relied solely on fixed, hand-designed weights and thresholds.

Activation Functions and Network Topologies. While a single perceptron can weigh evidence to make a simple (linear) decision, its power is amplified when multiple perceptrons are organized into a *network* of layers. In such a network, the first layer makes decisions based on the raw input evidence. The outputs of this layer are then fed as inputs to a second layer, which can, in turn, make decisions at a more complex and abstract level. By stacking multiple layers, the network can engage in highly sophisticated decision-making.

From an abstract perspective, a neural network can be represented as a directed acyclic graph composed of interconnected nodes, or *neurons*, inspired by their biological counterparts. Each neuron receives a set of input signals, each associated with a specific weight, and produces an output signal as a learned response. The graph structure enables data propagation from input neurons to output neurons through a series of weighted connections that modulate signal strength at each stage.

Despite its foundational importance, the perceptron is a purely linear classifier: its decision boundary is always a hyperplane, which limits it to solving only *linearly separable* problems. This critical shortcoming motivated the introduction of *activation functions*, non-linear transformations applied to the weighted sum before producing the neuron's output.

Formally, a modern neuron computes:

$$a = f\big(\mathbf{w} \cdot \mathbf{x} + b\big), \tag{2.3}$$

where $f(\cdot)$ is a non-linear activation function and a is the neuron's *activation*, which is subsequently transmitted to other neurons in the network. One of the earliest replacements for the perceptron's step function was the *sigmoid* function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{2.4}$$

which maps any real-valued input to the interval $(0,\, 1)$ and is smooth everywhere, enabling gradient-based learning algorithms such as backpropagation. Currently, the Rectified Linear Unit (ReLU) [27] is the most widely used activation function:

$$\mathrm{ReLU}(z) = \max(0,\, z). \tag{2.5}$$

ReLU introduces non-linearity at the origin and has proven to train faster than smooth alternatives on deep architectures due to its non-saturating gradient for $z > 0$. With

non-linearity, each successive layer can build on the features identified by the previous one to model increasingly complex patterns.

Deep Neural Networks and CNNs. When a neural network is constructed with a substantial number of layers stacked sequentially, it is referred to as a **Deep Neural Network** (DNN). This "depth" allows the network to learn a hierarchical representation of data: early layers extract low-level features (such as edges or textures in images), while deeper layers compose them into higher-level abstractions (such as objects or semantic concepts).

Computationally, each neuron in a DNN performs a weighted summation of its inputs, an operation that primarily consists of *multiply-accumulate* (MAC) computations, followed by the application of an activation function. The output activation is then transmitted to connected neurons in the next layer, often in a multicast (one-to-many) manner. In practice, DNNs are composed of various types of specialized layers, each implementing a specific mathematical function or *computational kernel*. Among the most prominent are **Convolutional Neural Networks (CNNs)**, which are specifically designed to process grid-like data such as images. CNNs rely on convolutional layers where small, learnable filters slide across input feature maps to extract spatial hierarchies of features. While this localized connectivity and weight sharing drastically reduce the number of distinct parameters compared to fully connected layers, they still generate an enormous number of MAC operations. Because of this massive computational density and the sheer volume of data they process, the efficient execution of these workloads has become a first-class engineering challenge, as will be discussed in Section 2.3.

2.2. DNN Workloads and Notation

Having established what a DNN is in Section 2.1, we formalize the *computational workload* that each layer imposes on hardware.

2.2.1. A Layer as a Loop Nest

Every CNN layer of a DNN can be decomposed into one or more *multiply-accumulate* (MAC) operations applied to three tensors: an *input activation* tensor, a *weight* (or filter) tensor, and an *output activation* tensor. A bias term, when present, is absorbed into the output initialization and does not affect the loop-nest structure.

The MAC is associative and commutative, the order in which the individual multipli-

cations and accumulations are carried out is irrelevant to the final numerical result, it means the computation can be expressed as a set of perfectly nested loops iterating over the various tensor dimensions, and one is free to tile, reorder, and parallelize these loops without altering correctness of the MAC.

The variable set that describes what iterates in this loop nest is the **dimensions** D of the workload to compute. The cardinality of D depends on the specific workload. For instance, for a fused 2D convolutional workload, Table 2.1 describes what each variable semantically represents.

Table 2.1: Semantic representation of loop nest dimensions for a 2D fused convolutional workload.

Dimension	Semantic Representation
C	Filter depth for each filter group / Input depth
R	Filter height
S	Filter width
M	Filter number of filter groups / Output depth
P	Output height
Q	Output width

Example: one-dimensional convolution. A 1D convolution that maps an input with C channels and spatial extent H to an output with M channels and spatial extent P, using filters of size R, is described by these MACs formula:

$$\texttt{Output}[m,\, p] \;=\; \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \texttt{Input}[c,\, p{+}r] \;\times\; \texttt{Filter}[m,\, c,\, r], \tag{2.6}$$

where $m \in [0,\, M)$ and $p \in [0,\, P)$. Each output element requires $C \times R$ MAC operations, since there are $M \times P$ the total MACs are $M \times P \times C \times R$.

This expression can be read directly as a four-deep loop nest:

```
for m in [0, M):
  for p in [0, P):
    for c in [0, C):
      for r in [0, R):
        Out[m][p] += In[c][p+r] * W[m][c][r]
```

The four loop dimensions, M, P, C, R, constitute the *iteration space* of the computation. These loops can be freely reordered (all 4! permutations produce the same output) and each loop can be tiled (split into an outer and an inner sub-loop in different memories)

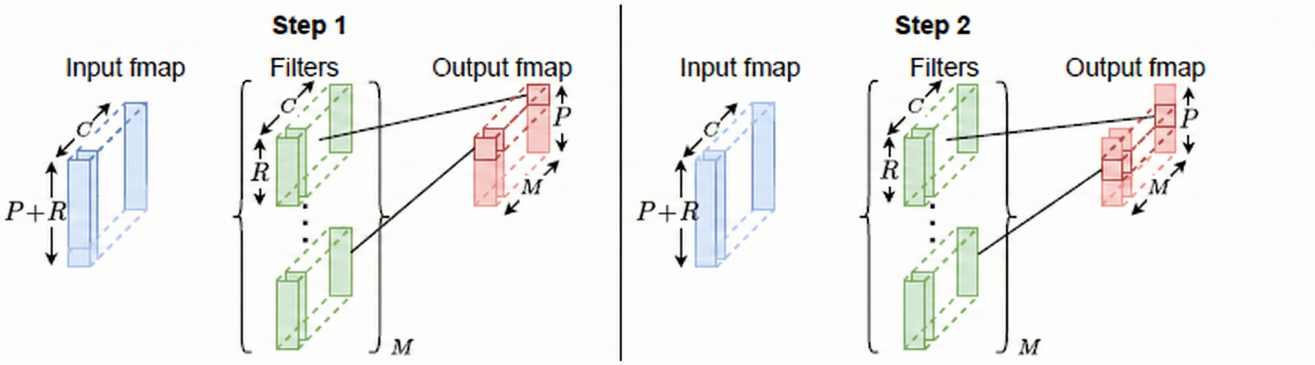

Figure 2.1: 1D convolution, P as the innermost loop

to create smaller sub-dimensions that fit in limited on-chip memories. This flexibility is the foundation of all mapping strategies discussed in Section 2.4.

Extension to 2D convolutions. The standard 2D convolution, as used in Convolutional Neural Networks (CNNs), adds a second spatial dimension and a batch dimension, leading to a seven-deep loop nest:

$$\texttt{O}[n,\, m,\, p,\, q] = \sum_{c} \sum_{r} \sum_{s} \texttt{I}[n,\, c,\, p{+}r,\, q{+}s] \;\times\; \texttt{W}[m,\, c,\, r,\, s], \tag{2.7}$$

with dimensions N (batch), M (output channels), P and Q (output spatial), C (input channels), and R, S (filter spatial).

2.2.2. Coupling and Shape: the Relationship between Dimensions and Tensors or Dimensions and Size

Each loop dimension in the loop-nest participates in the indexing of one or more of the three operand tensors. The precise binding between dimensions and tensors, which is called ***coupling***, fully determines what data each iteration reads and writes, and therefore what reuse opportunities exist.

A dimension d is said to be **coupled** to an operand if d appears in that operand's index expression (i.e., changing d changes the element accessed). Conversely, d is **orthogonal** to an operand if d does not appear in any of its index expressions; the operand's tile remains constant as d iterates, enabling complete *stationarity* of that operand.

Considering the 2D convolution of Eq. (2.7)., table 2.2 shows the coupling for each operand. Every dimension is coupled to exactly two of the three operands and orthogonal

to one, creating a clean three-way partition that directly maps onto the classical stationary dataflows (weight-stationary, input-stationary, output-stationary), as discussed in Section 2.4.

Table 2.2: Coupling of the standard 2D convolution. A ✓ indicates the dimension is coupled to (indexes) the operand; a – indicates orthogonality.

Dimension	Input	Weight	Output
M (out channels)	–	✓	✓
C (in channels)	✓	✓	–
R (filter height)	✓	✓	–
S (filter width)	✓	✓	–
P (out height)	✓	–	✓
Q (out width)	✓	–	✓
N (batch)	✓	–	✓

While the coupling map the relation between dimension and tensor, the ***shape*** is a dictionary mapping every dimension name to its size. This dictionary must be compatible with the coupling, for instance I cannot use a shape defining size of dimensions of a workload 2-D and use a coupling of a workload 1-D.

2.2.3. Normal and Affine Dimensions

Not all loop dimensions interact with tensor indices in the same way. We distinguish two fundamental categories:

A dimension d is **normal** if it appears alone as a tensor index for every operand it is coupled to. A normal dimension indexes each coupled operand independently: changing d selects a completely *different* element along that operand's corresponding axis.

A dimension d is **affine** if, for at least one operand, it participates in a *sum of indices* used to access a tensor dimension.

Two dimensional convolution coupling. In the two dimensional convolution of Eq. (2.7), the input is indexed along its height by $p + r$. Therefore:

- M and C are **normal** dimensions: M alone indexes the weight and output channel axes, C alone indexes the input and weight channel axes.
- R is **affine**: it appears in the sum $p + r$ that indexes the input's height. Likewise S is affine via $q + s$.

- P is also **affine**: it appears in the same sum $p + r$. Q is affine via $q + s$.

The normal/affine distinction has deep consequences for mapping:

1. **Tile-size computation.** When a normal dimension d is tiled to size t_d at some level, the operand tile along that axis is exactly t_d elements. For an affine pair such as (P, R), the input tile along the corresponding spatial axis is $t_P + t_R - 1$ (assuming unit stride), because consecutive output tiles *overlap* in their input requirements.

2. **Halo reuse.** When the innermost iterated dimension at a memory level is affine for an operand, consecutive iterations produce overlapping input tiles. Only the non-overlapping slice must be computed, while the remaining data can be reused from the immediately precedent iteration. This *halo reuse* is unique to affine dimensions and does not arise with normal ones.

3. **Map-space counting.** Only normal dimensions contribute independent tiling factors to the map-space cardinality (see Section 2.5). Affine dimensions affect tile sizes but are not independently looped.

2.2.4. Extended Einsum Notation

To express the structure of tensor computations more concisely, we adopts the **extended Einsum notation** (EEN) [1], which augments the standard Einstein summation convention with three features particularly suited to DNN workloads:

1. **Named ranks.** Each tensor dimension is given an explicit name (uppercase letter), and the variable iterating over it is denoted by the corresponding lowercase letter.

2. **Shape tags.** Superscripts on each tensor specify the size of every rank, making the shapes explicit in the notation.

3. **Affine subscripts.** Index expressions may contain sums of variables (e.g., $p + r$), directly encoding sliding-window and strided access patterns.

Using the extended Einsum notation, the 1D convolution from Eq. (2.6) can be rewritten as:

$$\text{Output}^{M,P}_{m,p} = \sum_{c,r} \text{Input}^{C,H}_{c,p+r} \times \text{Filter}^{M,C,R}_{m,c,r}. \tag{2.8}$$

The superscripts define the shapes of the tensors (e.g., the input has C channels and spatial extent $H = P + R - 1$, with stride equal to 1). The subscript $p + r$ makes the sliding-window access pattern immediately visible: as p advances, the accessed region in the input shifts by one, overlapping with the previous region in $R - 1$ elements. Padding

is handled implicitly: any access with indices outside the valid bounds of a rank is treated as zero.

The 2D convolution of Eq. (2.7) becomes:

$$\mathrm{O}^{N,M,P,Q}_{n,m,p,q} = \sum_{c,r,s} \mathrm{I}^{N,C,H,W}_{n,\,c,\,p+r,\,q+s} \times \mathrm{W}^{M,C,R,S}_{m,c,r,s}\,. \tag{2.9}$$

The extended Einsum notation is a compact specification that maps directly to the loop-nest representation of Section 2.2.1. Every rank in the notation corresponds to a loop dimension, every affine subscript encodes a coupled (non-orthogonal) relationship between two dimensions, and the implicit summation convention ($\sum_{c,r,s}$) identifies the contraction (reduction) dimensions. This one-to-one correspondence between the algebraic notation and the loop-nest representation makes EEN a natural interface for specifying workloads to mapping frameworks.

2.2.5. Activation-Dominant and Weight-Dominant Workloads

The coupling structure and dimension sizes of a layer determine which operand generates the most data traffic, for each layer. This distinction is important for selecting mapping strategies.

A layer is **activation-dominant** when the total size of its activation tensors (input and output feature maps) exceeds the total size of its weight tensor. This is typical of the early layers of a CNN, where spatial dimensions (P, Q) are large and the number of channels (M, C) is small.

A layer is **weight-dominant** when its weight tensor is larger than its activation tensors. This occurs in later layers of deep networks, where spatial dimensions have been reduced (through pooling or strided convolution) and the channel count has grown substantially.

Implications for layer fusion. Layer fusion primarily targets the elimination of intermediate activation traffic between consecutive layers (see Section 2.7). Its benefit is therefore most pronounced in activation-dominant layers, where the suppressed DRAM transfers of feature maps constitute a large fraction of total data movement. In weight-dominant layers, the potential savings from fusion are smaller in relative terms, since the dominant traffic component, the weights, is not reduced by fusion. This trade-off is examined quantitatively in the experimental evaluation (Chapter 7).

2.3. Hardware Acceleration for Deep Learning

The computational kernels central to DNNs, primarily general matrix multiplications and convolutions, are mathematically straightforward. However, because they operate on massive, high-dimensional tensors and are executed hundreds or thousands of times within a single network pass, they impose exceptionally high computational and memory bandwidth demands. Fortunately, the inherent simplicity of these operations is also their greatest asset. DNN workloads are highly *uniform*, exhibit highly *predictable* data dependencies and memory access patterns, and offer *abundant data-level parallelism*. This structural regularity makes them ideal candidates for specialized hardware acceleration, unlocking numerous opportunities for execution speedups and aggressive energy-saving optimizations that general-purpose processors simply cannot achieve.

2.3.1. From CPUs to Spatial Architectures

General-purpose CPUs can execute DNN workloads through their Single-Instruction, Multiple-Data (SIMD) vector units, but they are fundamentally constrained by a control-heavy micro-architecture designed for diverse, irregular code. GPUs improve on CPUs by providing thousands of lightweight threads operating in a Single-Instruction, Multiple-Thread (SIMT) model, achieving higher throughput on data-parallel workloads. However, GPUs still rely on general-purpose cores and a cache-based memory hierarchy that does not explicitly exploit the reuse patterns specific to DNN computations.

Spatial Architectures (SAs) represent a fundamentally different approach: instead of relying on shared, general-purpose cores, they employ a multidimensional array of dedicated *Processing Elements* (PEs) paired with an explicit on-chip memory hierarchy (registers, scratchpads) engineered to exploit the regular data dependencies and access patterns inherent to DNN kernels. Each PE is a simple functional unit executing *multiply-accumulate* (MAC) operations, and the architecture is designed such that data is passed between PEs and memories through controlled, predictable interconnects, maximizing *data reuse* and minimizing costly accesses to off-chip DRAM.

This specialized design philosophy has led to substantial improvements in both performance and energy efficiency. For example, Google's TPUv1 [13] achieved a 15–30× increase in computational throughput and a 30–80× improvement in energy efficiency compared to contemporary CPUs and GPUs executing the same inference workloads.

2.3.2. A Formal Abstraction of Spatial Architectures

Given the vast diversity of existing spatial architectures, systematically evaluating their performance requires a formalized, mathematical representation. To achieve this, our work adopts a specific architectural abstraction, foundational to analytical frameworks, that models hardware as a strict temporally sequential hierarchy. In this paradigm, any spatial accelerator is decomposed into three distinct *level types*, arranged from the outermost (closest to off-chip memory) to the innermost (closest to the computation):

1. **Memory levels** store tiles of one or more operands (inputs, weights, outputs) and are characterized by their *capacity* (in number of bytes), *bandwidth* (bytes per clock cycle), and per-access *energy cost* (read and write, in pJ). A memory level may *bypass* certain operands (input, weight, output), meaning it does not store them and delegates their handling to adjacent levels. The outermost level of any architecture must be a memory level, typically representing DRAM or another form of off-chip storage.

2. **Fanout (spatial) levels** model the physical replication of all subsequent levels into a multidimensional mesh of instances. A fanout level is defined by its *mesh size* (i.e., the number of parallel instances it creates) and the set of loop dimensions that can be spatially unrolled across those instances. Fanout levels are the mechanism through which spatial architectures achieve parallelism: each instance in the mesh executes the same inner loop nest concurrently, but on different tiles of data.

   Fanout levels additionally specify hardware support for *spatial multicast* (one read from an outer memory feeds all instances requiring the same data) and *spatial reduction* (partial results from multiple instances are accumulated into a single write to an outer memory). When multicast or reduction support is absent, each instance must independently fetch its own copy of shared data, significantly increasing bandwidth pressure on outer memories.

3. **Compute levels** represent the functional units (PEs) that perform the MAC operations. A compute level is defined by its *energy per MAC* and the number of *clock cycles per MAC*. The compute level must be the innermost level in the hierarchy and appears exactly once.

Fanout and compute levels are collectively referred to as **spatial levels**, because they create multiple concurrent instances of the computation. The key insight of this abstraction is that any level, whether memory or spatial, can be described through the same lens: each level holds a copy of the loop nest with its own *factors* (the number of iterations

unfolding at that level for each dimension) and its own *dataflow* (the order of those loops). Combined, the factors and dataflows of all levels determine the complete mapping (see Section 2.4).

2.3.3. Data Reuse in Spatial Architectures

The primary advantage of spatial architectures over general-purpose processors lies in their ability to systematically exploit *data reuse*, the phenomenon whereby a single data element is accessed multiple times during a computation, and only the first access need be serviced by an expensive outer memory. Three forms of reuse are central to the architecture model:

- **Temporal reuse** occurs when a data element resides in a memory level and is read or updated multiple times by consecutive iterations at that level, without being evicted back to outer memory. *Stationarity* (weight-stationary, input-stationary, output-stationary) is a specific form of temporal reuse: an operand is stationary at a given level when the innermost loops at that level iterate over dimensions *orthogonal* to it (see Section 2.2.2).
- **Spatial multicast reuse** occurs across the instances of a fanout level. When a fanout spatially unrolls a dimension that is orthogonal to an operand, all instances require the same tile of that operand. With hardware multicast support, a single read from the outer memory is broadcast to all instances, amortizing the access cost.
- **Spatial reduction reuse** is the dual of multicast: when a fanout spatially unrolls a dimension that is coupled to an output (i.e., multiple instances produce partial sums for the same output tile), hardware reduction support allows those partial sums to be accumulated and written back as a single result.

Maximizing reuse is the central objective of the mapping problem: accessing off-chip DRAM can cost hundreds of times the energy of a PE-local register read (e.g., 100/200 pJ per byte vs. ∼0.1 pJ per byte [6]), making data movement the dominant contributor to total energy unless reuse is carefully orchestrated.

2.4. The Mapping Problem

Given a DNN workload expressed as a loop nest (Section 2.2) and a spatial architecture described as a hierarchy of levels (Section 2.3), a **mapping** is a complete specification of how the computation is partitioned, scheduled, and assigned to the hardware. Concretely, a mapping answers three questions:

1. **Where** does each operand tile reside in the memory hierarchy?
2. **When** and **how** does data move between levels, and which multicast or reduction opportunities are exploited?
3. **Which PE** computes **which MAC**, and in what temporal order?

The set of all mappings that satisfy the resource and user constraints of a given workload–architecture pair is called the **map-space** $\mathcal{MS}$. The **mapping problem** is to find a mapping $m^* \in \mathcal{MS}$ that minimizes a target metric, typically energy, latency, or the energy-delay product (EDP).

This optimization is difficult for two reasons. First, the map-space is combinatorial large (Section 2.5 quantifies this). Second, energy and latency are dominated by *data movement* rather than by computation: accessing off-chip DRAM costs hundreds of times the energy of a PE-local register read, so the objective function is highly sensitive to how effectively a mapping exploits data reuse.

2.4.1. Three Degrees of Freedom

Every mapping configuration within the design space is fully characterized by three orthogonal choices, which are evaluated *independently at each level* of the architectural hierarchy.

Tiling

To accommodate massive neural network tensors within the strictly limited capacities of on-chip SRAMs, the overall execution loop nest must be *tiled*. This means each loop dimension d, with a total extent of $|d|$, is split into smaller sub-ranges distributed across the memory hierarchy.

Formally, the total extent of each dimension is factorized, and these individual factors are assigned to specific memory levels. The product of the factors assigned to a given level, combined with all factors assigned to the levels below it, dictates the *tile size* at that level, the number of elements along dimension d that the hardware must concurrently store to process an iteration.

Tiling choices directly dictate the storage requirements at each level. A mapping is considered *physically valid* only if the combined size of all required operand tiles fits strictly within the hardware capacity of the assigned memory level.

Example. Consider a loop dimension $M = 128 = 2^7$ mapped onto a three-level hierarchy

(DRAM → GlobalBuffer → Register). A valid tiling strategy might assign a factor of 2^3 to the DRAM, 2^2 to the GlobalBuffer, and 2^2 to the Register, resulting in local tile sizes of 128, 16, and 4, respectively. Consequently, each 4-element register tile is temporally reused across 4 innermost iterations before a new tile must be fetched from the GlobalBuffer, while the 16-element GlobalBuffer tile is reused across 8 iterations before triggering a DRAM access.

Parallelism Strategy

Just as tiling distributes temporal iterations across memory levels, the *parallelism strategy* distributes spatial iterations across the hardware's compute array. Factors of a dimension can be assigned to *fanout levels*, determining the number of hardware instances (PEs) executing in parallel along that dimension.

This spatial unrolling tightly interacts with data reuse through hardware *multicast* and *reduction* networks:

- **Multicast:** When a fanout level unrolls a dimension that is orthogonal to a specific operand, all PEs require the exact same data tile. If the architecture supports spatial multicast, a single read from the outer memory is broadcast to all PEs, effectively multiplying the data reuse by the fanout mesh size.
- **Reduction:** Conversely, when the unrolled dimension is coupled to the output, multiple PEs simultaneously produce partial sums for the exact same output index. Hardware spatial reduction support allows these partial sums to be accumulated automatically in transit, resulting in a single write operation to the outer memory.

Furthermore, when the unrolled dimension has an *affine* relationship to an operand, adjacent PEs access *overlapping* (but not identical) input tiles due to the sliding-window nature of convolutions. If the architecture features *selective multicast*, it can fetch the union of all required tiles once and route the correct overlapping sub-tiles to the appropriate PEs. Without this support, every PE must independently fetch its entire footprint, severely multiplying the external bandwidth demand.

Loop Ordering (Dataflow)

At each memory level, the loops assigned to that level can be *permuted* without altering the mathematical correctness of the convolution. This chosen permutation constitutes the level's **dataflow schedule** and serves as the primary knob to control which temporal reuse mechanisms are activated.

The core principle is that operands remaining *constant* across the innermost loop iterations are held stationary in the local memory, effectively bypassing redundant read operations. Thus, the innermost loops dictate the stationarity of the system. The interaction between loop ordering and operand coupling yields three distinct reuse patterns:

1. **Stationarity (Temporal Reuse).** When the innermost iterated dimension at a memory level is *orthogonal* to an operand (cf. Definition 2.2.2), that operand's index does not change across consecutive iterations. The tile is perfectly reused, making the operand *stationary*. In a standard 2D convolution, each dimension is orthogonal to exactly one of the three operands (Table 2.2), meaning at most one operand can be made purely stationary per memory level. This mathematical constraint yields the three classical architectural dataflows:
    - *Weight-Stationary (WS):* Innermost loops iterate over dimensions orthogonal to the weights (e.g., P, Q), retaining the weight tile in local registers.
    - *Input-Stationary (IS):* Innermost loops iterate over dimensions orthogonal to the inputs (e.g., M), retaining the input tile.
    - *Output-Stationary (OS):* Innermost loops iterate over dimensions orthogonal to the outputs (e.g., C, R, S), retaining the partial-sum tile locally to accumulate results before a final write-back.
2. **Halo Reuse (Temporal, Affine).** When the innermost iterated dimension is *affine* with respect to an operand, consecutive iterations require tiles that heavily *overlap* due to the sliding-window access pattern.The magnitude of this reused "halo" depends strictly on the affine coefficients (strides) and the current tile sizes. Crucially, halo reuse can only be exploited if none of the dimensions involved in the affine index sum are spatially parallelized at lower levels; otherwise, the overlapping data would be dispersed across different physical PEs, breaking the temporal reuse chain.
3. **Spatial Reuse.** On a *fanout* level, the temporal loop ordering is irrelevant; only the *choice of which dimensions are parallelized* matters. Parallelizing a dimension orthogonal to an operand triggers *multicast*, while parallelizing a dimension coupled to the output triggers *reduction*. For affine dimensions, overlapping tiles across spatial instances unlock complex selective multicast or reduction opportunities, provided the underlying network supports them.

2.4.2. Interaction of the Three Degrees of Freedom

Tiling, parallelism, and loop ordering are not independent in their effect on performance:

- **Tiling + parallelism** jointly determine what data must be where and when. The tile sizes at each level define the storage footprint, while the parallelism strategy determines how many copies of inner levels concurrently share an outer memory's bandwidth.
- **Loop ordering** then schedules the temporal sequence of data transfers and decides which reuse mechanism fires at each memory level. A specific loop order can only exploit stationarity or halo reuse if the tiling has allocated enough iterations to the relevant dimension at that level.

At any memory level, the permutation of the loops determines which reuse fires:

1. *Stationarity* is locked by the set of innermost loops that are orthogonal to an operand.
2. *Halo reuse* depends on the position of the first *coupled* loop and whether it involves an affine dimension.
3. On *spatial* levels, loop order does not change behavior; only the choice of parallelized dimensions matters.

This interplay means that evaluating a mapping requires considering all three decisions simultaneously, which is precisely why the mapping problem is so challenging: the search space is the Cartesian product of all tiling configurations, all parallelism assignments, and all loop permutations at every level. Section 2.5 quantifies the resulting combinatorial explosion.

2.5. Why the Map-Space Explodes

Section 2.4 introduced the mapping problem and its three degrees of freedom. This section demonstrates that even modest workloads on relatively simple architectures produce map-spaces of astronomical size, motivating the need for analytical cost models.

Two observations underpin the explosion:

- **Permutations per memory level.** Loop *ordering* at memory levels matters because it determines which operand is stationary, whether halo reuse is available, and in what sequence transfers occur. In contrast, on spatial levels loop order does not

change behavior: only the chosen *parallelized dimensions* matter (Section 2.4.1).

- **Independent tiling choices.** For each *normal* dimension $d \in D$ (cf. Definition 2.2.3), we decompose its extent into prime factors and distribute the resulting copies across the L levels of the architecture. These choices are independent across distinct primes and across dimensions.

2.5.1. Formal Setup and Notation

Let L denote the total number of levels in the spatial architecture and $L_m \leq L$ the number of *memory* levels. Let D be the set of normal kernel dimensions (affine dimensions affect tile *sizes* but are not independently looped; see Section 2.2.3). For each dimension $d \in D$, write the prime factorization of its extent as

$$|d| \;=\; \prod_{p} p^{v_p(d)},$$

where $v_p(d)$ is the p-adic valuation of $|d|$ (i.e., the exponent of prime p in the factorization).

To understand the sheer scale of the mapping problem, it is useful to establish an upper bound on the total number of possible scheduling configurations. By temporarily ignoring physical hardware validity constraints (such as mappings that exceed SRAM or register capacities), we can evaluate the theoretical size of the map space. This unconstrained design space is driven by two independent combinatorial degrees of freedom: the permutation of loops at each memory level and the allocation of prime factors across the hardware hierarchy.

Loop Permutations. On each memory level, we may arbitrarily permute the $|D|$ loops to define the local dataflow. This yields $|D|!$ ordering choices per memory level, resulting in:

$$(|D|!)^{L_m}$$

total permutations across the architecture. Spatial levels do not contribute to this term because, as established in Section 2.4.1, temporal loop order on a purely spatial fanout level is irrelevant.

Factor Allocations. Independent of the loop ordering, the workload dimensions must be tiled. For a fixed dimension d and a fixed prime p, we must distribute $v_p(d)$ indistinguishable copies of p across the L *labeled* levels of the architecture. Since levels may receive zero copies, and assigning factors to different levels produces physically different

tile sizes, the order of assignment strictly matters. The number of such allocations is mathematically equivalent to the number of weak compositions of $v_p(d)$ into L parts:

$$\binom{v_p(d) + L - 1}{L - 1}.$$

Because these allocations are independent across distinct primes $p \mid d$ and across distinct dimensions $d \in D$, the *total* number of possible factor allocations is the product over all dimensions and their prime factors:

$$\prod_{d \in D} \prod_{p \mid d} \binom{v_p(d) + L - 1}{L - 1}.$$

2.5.2. Map-Space Cardinality

Combining permutations and factor allocations yields the following expression for the (valid + invalid) map-space size [24]:

$$|\mathcal{MS}| = \underbrace{(|D|!)^{L_m}}_{\text{loop permutations}} \cdot \underbrace{\prod_{d \in D} \prod_{p \mid d} \binom{v_p(d) + L - 1}{L - 1}}_{\text{factor allocations}}. \tag{2.10}$$

Growth with architecture complexity. The permutation term scales *exponentially* in L_m (with base $|D|!$), while the factor-allocation term grows *polynomially* in L, with the exponent equal to the total number of prime copies $\sum_{d \in D} \sum_{p \mid d} v_p(d)$, which itself increases with $|D|$ and the loop extents.

2.5.3. Worked Example

Consider a standard 2D convolution with $|D| = 6$ normal loops (e.g., M, C, P, Q, R, S) mapped onto an architecture with $L = 5$ levels, of which $L_m = 4$ are memory levels.

Permutations.

$$\text{Permutations} = (6!)^4 = 720^4 \approx 2.68 \times 10^{11}.$$

Factor allocations. Given each dimension has, in total, the same five prime-factor copies to distribute across the five levels.

Two boundary cases illustrate the range:

1. **All five copies on the same prime** (e.g., $v_p(d) = 5$, unique prime repeated 5 times; for instance this happen with size 32): for each dimension $\binom{5+5-1}{5-1} = \binom{9}{4} = 126$, so across six dimensions $126^6 \approx 1.07 \times 10^{13}$.
2. **Five distinct primes, one copy each** ($v_p(d) = 1$ for five primes; for instance this happen with size 2310): per dimension $\left(\binom{1+5-1}{5-1}\right)^5 = 5^5 = 3125$, so across six dimensions $3125^6 \approx 3.05 \times 10^{20}$.

Total. Multiplying by permutations, the map-space size ranges between

$$\underbrace{720^4 \cdot 126^6}_{\approx 1.1\times10^{24}} \quad \text{and} \quad \underbrace{720^4 \cdot 3125^6}_{\approx 2.5\times10^{32}}.$$

Even the conservative end of this range *far* exceeds 10^{24}. These numbers represent *all* mappings, including invalid ones; nevertheless, they demonstrate that exhaustive enumeration is infeasible.

Practical implications. Across realistic map-spaces, EDP can vary by up to $10^4\times$ between the best and worst valid mappings, and near-optimal points are sparsely distributed [14]. Small random sampling budgets are therefore unlikely to discover good solutions without *structural pruning*, heuristics that eliminate provably sub-optimal or redundant regions of the search space before evaluation. This motivates the use of analytical cost models that can evaluate candidate mappings cheaply (Section 2.6) and mapper algorithms that exploit the structure of the map-space to navigate it efficiently.

2.6. Cost Modeling

Sections 2.4 and 2.5 established that the mapping problem requires choosing one mapping out of a combinatorially large space and that the optimal choice depends on the *cost* of each mapping (in terms of Energy-Delay-Product, Energy or Latency. This section formalizes how that cost is estimated.

A **cost model** is a function that takes an architecture description and a mapping and returns the cost without actually fabricating the chip or running the workload on real hardware. Two fundamentally different strategies exist.

- **Simulation-based** models (e.g. cycle-accurate simulators) execute the data move-

ment and compute operations on a software model of the hardware, tracking every memory access and pipeline event. They can be highly accurate, but their runtime grows with the number of operations in the workload, making them impractical when millions of mappings need to be evaluated.

- **Analytical** models compute the cost in closed form (or near–closed form) from the mapping's tiling factors, loop ordering, and the architecture's parameters. Their runtime is independent of the workload size, enabling rapid exploration of large map spaces. The trade-off is that simplifications, such as assuming perfect double-buffering or ignoring interconnect congestion, introduce approximation error.

Because map-space exploration (Section 2.5) requires evaluating a cost function for a very large number of candidate mappings, analytical models are the predominant choice in modern DNN mapping frameworks [15, 19, 21]. The remainder of this section describes the three building blocks of such models: the *memory-access count*, the *energy model*, and the *latency model*, together with the compound metrics derived from them.

2.6.1. Memory-Operation Count

The starting point of any analytical cost model is to determine how many *memory operations* (MOPs) each level of the hierarchy performs for a given mapping. A memory operation is a single scalar read or write from/to a storage level; the total count depends on the tiling, loop ordering, and bypass configuration of the mapping (Sections 2.4.1–2.4.1).

Consider a memory level ℓ that stores operands $\mathcal{O} \subseteq \{\text{in}, \text{w}, \text{out}\}$ (the remaining operands bypass the level). At each iteration of the loops mapped on ℓ, a *tile* of every stored operand must be read from ℓ and supplied to the levels below; partial results are written back from below. The total number of *reads* and *writes* at ℓ can be expressed as:

$$\text{reads}_\ell \;=\; \sum_{o \in \mathcal{O}} \frac{\text{tile}_o}{\text{reuse}_{o,\ell}} \;\cdot\; \prod_{k \in \mathcal{L}_\ell} f_k, \qquad \text{writes}_\ell \;=\; \text{fills}_\ell + \text{updates}_\ell, \tag{2.11}$$

where f_k are the tiling factors on level ℓ, the product ranges over the loops $\mathcal{L}_\ell$ mapped there, and $\text{reuse}_{o,\ell}$ captures temporal reuse from stationarity and halo overlap (Section 2.4.1). The writes are the sum of *fills.*, data moved into the level from above level, and *updates*, partial sums drained from below levels, reads and writes are tracked separately so that read and write bandwidths can be individually bounded.

Additionally, MOPs occurring at a fanout level (Section 2.3.2 are accounted for through *multicast* and *reduction*: when an operand is spatially reused across PEs, a single read

from the level above can serve multiple instances (multicast), and corresponding partial sums from multiple PEs may be aggregated before being written back (reduction).

2.6.2. Energy Model

Given the MOP counts, the total energy is obtained by weighting each access with the per-access energy of the corresponding storage level:

$$E = \underbrace{\sum_{\ell} \left(e_{\ell}^{\mathrm{rd}} \cdot \mathrm{reads}_{\ell} + e_{\ell}^{\mathrm{wr}} \cdot \mathrm{writes}_{\ell} \right)}_{\text{data-movement energy}} + \underbrace{\sum_{\ell} e_{\ell}^{\mathrm{leak}} \cdot L}_{\text{leakage}} + \underbrace{e^{\mathrm{MAC}} \cdot \mathrm{MACs}}_{\text{compute energy}}, \tag{2.12}$$

where e_{ℓ}^{rd} and e_{ℓ}^{wr} are the per-scalar read and write energies at level ℓ (in pJ), e_{ℓ}^{leak} is the leakage power per clock cycle, L is the total execution latency, and e^{MAC} is the energy of a single multiply-accumulate operation. The summation runs over all memory levels and the compute level.

The per-access energies are technology-dependent constants that characterize the hardware. They can be provided directly for each level (e.g. measured or taken from a datasheet) or estimated by an external tool like accelergy a widely used energy-estimation framework that queries technology-specific plug-ins, such as CACTI for SRAMs [18], to estimate the per-access energy and area of each component given its configuration (depth, width, technology node, number of ports, number of banks). Because the per-access energies are constants of the architecture and do *not* change across mappings, they can be queried once during architecture setup and then reused throughout exploration. Compute energy is fixed once the workload is determined: the total number of MACs is a property of the computation, not of how it is mapped. Since DRAM accesses are orders of magnitude more energy-expensive than register-file accesses or multiply accumulations ($\sim$100/200 pJ vs. $\sim$1 pJ vs. $\sim$0.2 pJ per byte accessed at 45 nm [10], [30]), data movement dominates the energy budget, and so minimizing the number of MOPs at the costliest levels is the primary objective of any mapping optimization.

2.6.3. Latency Model

The latency of a mapping is the total number of clock cycles from the first input read to the last output drain. In an analytical model, latency is driven by two bottlenecks at every memory level: the *compute bound* and the *bandwidth bound*.

Compute bound. If the available bandwidth is sufficient to keep the compute units fed, the latency at level ℓ equals the number of tiles processed times the latency of one single tile:

$$L_\ell^{\text{compute}} = c_{\text{tile}} \cdot \prod_{k \in \mathcal{L}_\ell} f_k, \tag{2.13}$$

where c_{tile} is the clock cycles required to process one tile (propagated upward from the compute level).

Bandwidth bound. If the aggregate data rate demanded by a level exceeds its read or write bandwidth B_ℓ, the level becomes a bottleneck and latency increases:

$$L_\ell^{\text{BW,rd}} = \frac{\text{reads}_\ell + \text{drains}_\ell}{B_\ell^{\text{rd}}}, \qquad L_\ell^{\text{BW,wr}} = \frac{\text{fills}_\ell + \text{updates}_\ell}{B_\ell^{\text{wr}}}. \tag{2.14}$$

The effective latency contributed by each level is then

$$L_\ell = \max\left(L_\ell^{\text{compute}},\ L_\ell^{\text{BW,rd}},\ L_\ell^{\text{BW,wr}}\right), \tag{2.15}$$

and the overall execution latency is the maximum across all levels:

$$L = \max_\ell\ L_\ell. \tag{2.16}$$

When a level's bandwidth is saturated, the excess demand translates into *stall cycles* that idle the compute units: $\Delta_\ell = L_\ell - L_\ell^{\text{compute}}$. A mapping with zero stall cycles at every level achieves full *compute utilization*; in practice, the slowest memory level dictates the overall throughput.

2.6.4. Compound Metrics

Energy and latency capture complementary aspects of a mapping's quality: a mapping may reduce energy by reusing data more but increase latency due to bandwidth stalls, or vice versa. To compare mappings along both axes simultaneously, compound metrics are commonly used.

Energy-Delay Product (EDP). The most widely adopted metric is the *energy-delay product*:

$$\text{EDP} = E \times L \qquad [\text{pJ} \cdot \text{cc}]. \tag{2.17}$$

Minimizing the EDP balances energy against latency: improving either dimension while not degrading the other yields a better mapping. EDP is used as the default optimization target by several frameworks, including Timeloop [21], and Factorflow [22] .

2.7. Layer Fusion

All preceding sections have implicitly assumed that only a *single* convolution is mapped at a time: the accelerator reads one layer's inputs and weights from DRAM, computes its outputs, writes them back to DRAM, and then moves on to the next layer. Under this **layer-by-layer** (LBL) execution model, every intermediate feature map must traverse the full memory hierarchy twice, once as the output of one layer and once as the input to the next. Since DRAM accesses dominate the energy budget (Section 7.1.1), these redundant round-trips constitute a major source of inefficiency in modern DNN inference.

Layer fusion is the strategy of co-executing two or more consecutive layers so that intermediate feature maps are produced *and consumed on-chip*, without being written to and so read from DRAM. This section introduces the concept, formalizes its mechanics, and discusses the open problems it raises.

2.7.1. Motivation

Consider a network of N consecutive layers $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{N-1}$, executed layer by layer. Let A_i denote the intermediate activation tensor between $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$. In LBL execution, each A_i is written to DRAM after $\mathcal{L}_i$ completes and read back when $\mathcal{L}_{i+1}$ starts, incurring

$$T_{\text{LBL}}^{\text{intermediate}} = 2 \sum_{i=0}^{N-2} |A_i| \tag{2.18}$$

DRAM accesses (one write plus one read per element). For deep networks with large spatial resolutions, this traffic can far exceed the traffic for inputs, outputs, and weights combined. For instance, even for a weight dominant workload (deep layers with many channels and small spatial dimensions) as VGG-16 [4], in the early layers of a the intermediate feature maps of a single 224×224 image occupy tens of megabytes, orders of magnitude larger than the corresponding filter weights.

When F consecutive layers ($\mathcal{L}_0$ through $\mathcal{L}_{F-1}$) are *fused*, all $F-1$ intermediate tensors $A_0, \ldots, A_{F-2}$ are kept on-chip and ideally never touch DRAM. The DRAM traffic for the

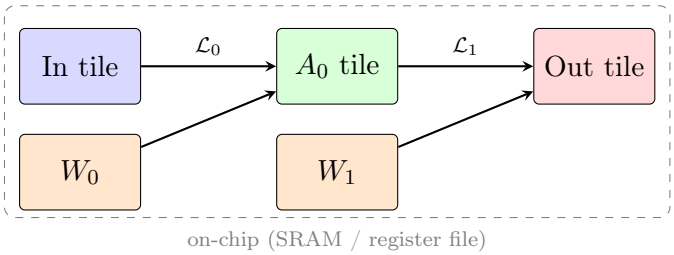

Figure 2.2: Fused execution of two consecutive layers. The intermediate activation A_0 is produced and consumed on-chip; only the network's input, weights, and final output cross the DRAM boundary.

fused group therefore reduces to

$$T_{\text{fused}} = \underbrace{|I|}_{\text{first layer's input}} + \underbrace{\sum_{i=0}^{F-1} |W_i|}_{\text{all weights}} + \underbrace{|O|}_{\text{last layer's output}}, \tag{2.19}$$

eliminating the intermediate traffic entirely.

2.7.2. Fused Tiling

Fusion does not simply mean running all layers in sequence inside a single invocation. The key insight is that the tiling of the outer (last) layer's output *dictates* the tiles required from every preceding layer through a chain of producer–consumer dependencies

Consider fusing two convolutional layers with filter sizes $R_0 \times S_0$ and $R_1 \times S_1$ respectively with strides assumed to be 1 for simplicity of the example. If the second layer requires an output tile of size $P \times Q$, it needs an intermediate tile of size $(P + R_1 - 1) \times (Q + S_1 - 1)$ from the first layer's output, (cf. the halo discussion in Section 2.4.1). To produce that intermediate tile, the first layer in turn needs an input tile of size

$$\big(P + R_1 - 1 + R_0 - 1\big) \times \big(Q + S_1 - 1 + S_0 - 1\big). \tag{2.20}$$

The tile sizes thus *cascade backwards* through the fused layers, and each intermediate tile is enlarged by its layer's corresponding halo. This cascading is precisely the phenomenon captured by *affine dimensions* (Definition 2.2.3 in Section 2.2.3): the sums of indices (e.g. $p + r_1$ mapping to intermediate spatial positions) create dependencies that propagate tile-size requirements across layers.

Memory-footprint implications. All intermediate tiles must fit simultaneously in an on-chip buffer, since they are produced and consumed between consecutive loop iterations without visiting DRAM. In the two-layer case the on-chip buffer must hold at least:

$$\text{footprint} \geq \underbrace{|I_{\text{tile}}|}_{\text{input tile}} + \underbrace{|A_{0,\text{tile}}|}_{\text{intermediate tile}} + \underbrace{|O_{\text{tile}}|}_{\text{output tile}} . \tag{2.21}$$

As more layers are fused, additional intermediate tiles must be stored concurrently, increasing the footprint. Correspondingly, the output tile size $P \times Q$ that fits within a given on-chip SRAM budget *decreases* with the number of fused layers, which can impact data reuse and, in turn, energy efficiency.

2.7.3. Inter-Layer vs. Intra-Layer Reuse

Two complementary forms of data reuse arise in fused execution:

Intra-layer reuse is the reuse discussed in Section 2.3.3: stationarity, halo overlap, and spatial multicast within a single layer. These mechanisms remain available inside each fused layer and are optimized through tiling, loop ordering, and spatial unrolling exactly as for a single-layer mapping.

Inter-layer reuse is unique to fusion. When intermediate activations reside on-chip, successive iterations of the *outer* (consuming) layer can reuse portions of the intermediate tile that were produced by a previous iteration of the *inner* (producing) layer, provided the halo regions overlap. The amount of inter-layer reuse depends on the relative tile sizes, filter dimensions, and loop ordering across layers.

The total on-chip reuse in a fused mapping is the combination of both forms. Tiling the intermediate tensor to maximize inter-layer reuse can starve intra-layer reuse within each individual layer, and vice versa.

2.7.4. Choosing Which Layers to Fuse

Not all layers benefit equally from fusion. The decision of which consecutive layers to group into a fused set depends on several factors:

- **On-chip memory budget.** Each additional fused layer increases the memory footprint (Equation (2.21)). If the on-chip buffer cannot accommodate the combined intermediate tiles, the output tile must be reduced, reducing data reuse and potentially increasing total energy, even with the minimum output tile can be possible to

don't have enough memory footprint.

- **Intermediate activation volume.** Fusion is most beneficial when the intermediate tensors are large, i.e. when the spatial resolution is high and the number of channels is moderate. This corresponds to the *activation-dominant* regime described in Section 2.2.5. Conversely, fusing layers in the *weight-dominant* regime (deep layers with many channels and small spatial dimensions) yields diminishing returns because the intermediate traffic is already small.
- **Diminishing returns.** Beyond a certain depth of fusion, the marginal DRAM traffic saved by adding one more layer decreases while the memory pressure on the on-chip buffer and the map-space size continue to grow. Finding the optimal *fusion depth*, how many layers to fuse together, is itself a combinatorial problem.

We focus on the *mapping* of an already-determined fusion set rather than on the selection of which layers to fuse. The analytical cost model and mapper developed in the following chapters accept any fusion depth and optimize the combined loop nest accordingly, enabling systematic evaluation of both shallow and deep fusion configurations.

3 | Fused-Layer Mapping Design Space

Chapter 2 has established the mapping problem for a *single* convolutional layer: given a loop nest and a spatial architecture, a mapping specifies how the computation is tiled, ordered, and distributed across the hardware (Section 2.4), and even for a single layer the resulting map-space can exceed 10^{24} candidate configurations (Section 2.5). Chapter 2 also has introduced layer fusion as a strategy to eliminate intermediate DRAM traffic (Section 2.7), showing how multiple layers are expressed as a single unified loop nest with five operand categories and cascading tile-size dependencies

This chapter bridges the gap between those two discussions. Layer fusion does not merely concatenate individual mappings; it introduces qualitatively new mapping decisions that have no counterpart in single-layer execution. Intermediate activations must be assigned to specific memory levels, inter-layer tile sizes must be chosen jointly, and bypass rules must be defined for each of the five operand types at every level of the hierarchy. These additional degrees of freedom enlarge the design space dramatically and create complex trade-offs between data reuse, on-chip memory capacity, and off-chip bandwidth.

3.1. From Single-Layer to Fused-Layer Mapping

3.1.1. Single-Layer Mapping Recap

In a standard single-layer mapping, the computation is fundamentally expressed as a single loop nest over $|D|$ dimensions (e.g., six dimensions for a 2D convolution, as detailed in Section 2.2.1). A mapping configuration is fully characterized by three independent decisions made at each level of the architectural hierarchy: *tiling* (how each dimension's extent is factored across the memory levels), *loop ordering* (the iteration sequence within each specific memory level), and *spatial parallelism* (which dimensions are unrolled across the PE array) (see Section 2.4.1). The analytical cost model evaluates the memory operations (MOPs), energy consumption, and cycle latency for each level independently

(Section 2.6). As previously established, the combinatorial explosion of valid tilings and orderings already makes exhaustive unguided search computationally infeasible for practical hardware architectures (Section 2.5).

3.1.2. The Architectural Shifts of Layer Fusion

Fusing F consecutive layers into a single, interleaved execution unit preserves the three foundational degrees of freedom (tiling, ordering, and parallelism) but alters the mapping problem in four fundamental ways.

Dimension Proliferation

In single-layer mapping, the loop nest contains a fixed number of dimensions ($|D|$). However, when F layers are mathematically unified into a fused loop nest, each layer contributes its own distinct set of per-layer dimensions. Consequently, the total dimensionality of the scheduling problem expands linearly with the fusion depth. Because the map-space cardinality depends combinatorially on both the *permutations* of dimensions at each memory level and the *factor allocations* of every dimension's prime factors across the hardware hierarchy (Equation 2.10), this dimensional proliferation triggers a massive, compounding explosion in the search space (quantified formally in Section 5.1.3).

Five Operand Categories

As introduced in Chapter 2, distinguishing intermediate activations from external memory traffic expands the tensor footprint into five distinct operand categories:

1. **Input**: The read-only tensor consumed exclusively by the first layer in the fused segment.
2. **Weights**: One independent set per layer ($W_0, \ldots, W_{F-1}$), each referenced strictly by its own layer's dimensional indices.
3. **Intermediate Output**: The localized feature map produced by every intermediate layer (e.g., layer i produces A_i).
4. **Intermediate Input**: The feature map consumed by every non-first layer (e.g., layer i reads A_{i-1}, which is the intermediate output of the preceding layer).
5. **Output**: The write-only tensor produced exclusively by the final layer.

Every memory level in the spatial architecture must explicitly define a bypass configuration for each of these five categories. These granular bypass rules directly control the

physical on-chip residency of intermediate activations, dictating the feasibility of the fused schedule.

Inter-Layer Tile-Size Dependencies

Fusion breaks the isolation of layer scheduling by introducing strict *cascading dependencies* between layers.

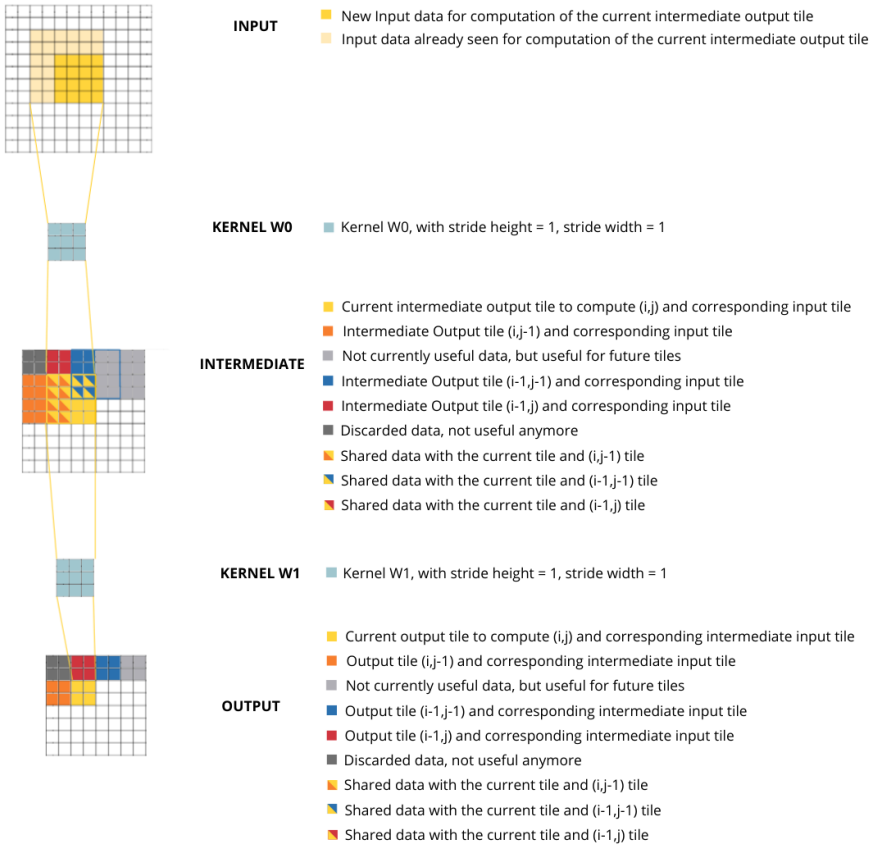

Figure 3.1: Example of output tile size 2x2 with stride 1

As discussed in Section 2.7.2, selecting an output spatial tile of size $P \times Q$ for the final layer mathematically forces every preceding layer to compute a progressively larger tile to account for overlapping filter halo. Accounting for independent strides along the spatial dimensions, (as shown 3.1 and in 3.2) in this constraint propagates backward according to:

$$\text{Layer } i \text{ (intermediate footprint)}: \quad \big(\text{stride}_h \cdot (P-1) + R\big) \times \big(\text{stride}_w \cdot (Q-1) + S\big)$$

This backward propagation dictates that inter-layer tile sizes are *strictly dependent*. A tiling choice made for the outermost layer rigidly constrains the required intermediate tile

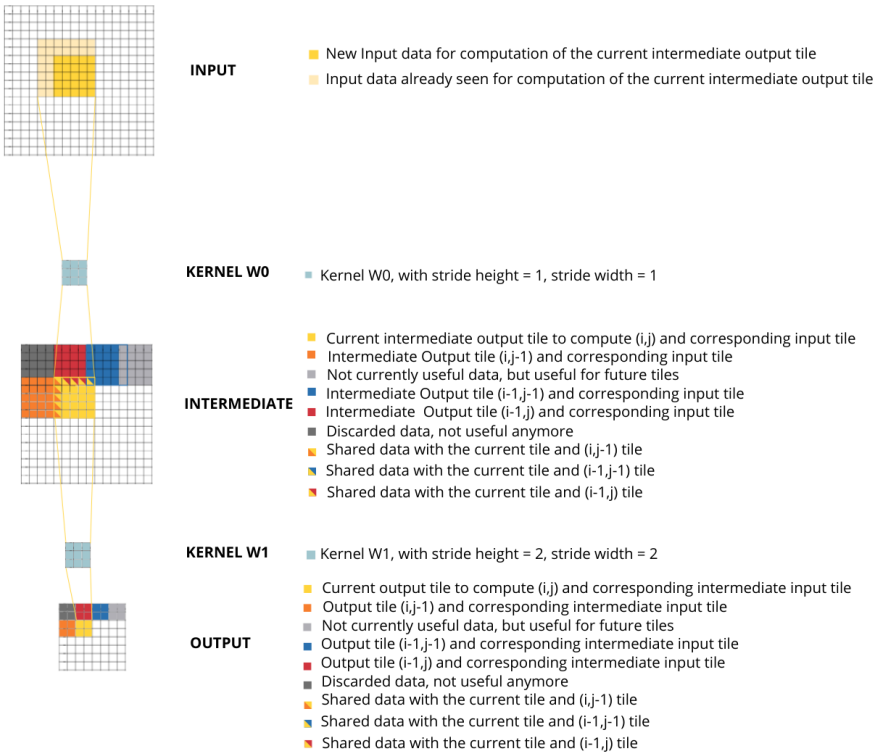

Figure 3.2: Example of output tile size 2x2 with stride 2

sizes at every preceding level of the computation.

The practical architectural consequence is severe: as fusion depth increases, intermediate tiles swell due to the cumulative halo effect. This inflates the on-chip SRAM footprint and restricts the maximum feasible output tile size that can fit within a fixed memory budget. This creates the fundamental architectural trade-off of fusion: deeper fusion topologies eliminate more external DRAM traffic, but exponentially tighten the on-chip capacity constraints for every participating layer.

Execution Models and Parallelism

In standard single-layer mapping, the question of *how layers interact in time and space* does not exist: each layer is completely mapped, executed, and flushed to DRAM before the next begins. Layer fusion, however, requires explicitly defining an *execution model* to manage the concurrent residency of multiple layers. Two principal models exist in the literature:

Sequential (Depth-First): for each spatial tile, all F layers are executed sequentially on the exact same shared PE array. The first layer produces an intermediate tile, which is buffered and immediately consumed by the second layer, continuing until

the final layer writes the completed output tile to memory. Only then does the accelerator advance to the next spatial spatial region. This model, adopted by architectures like DepFiN [10], does not physically partition the PE array.

Pipelined (Inter-Layer Parallel): different layers execute concurrently on physically distinct, spatially partitioned subsets of the PE array, functioning as continuous pipeline stages. While this inter-layer parallelism can increase throughput and reduce overall latency, it introduces severe control complexity and rigid hardware partitioning constraints.

The formalized analytical mapping framework developed and evaluated in this thesis strictly operates under the **Sequential (Depth-First)** execution model, evaluating the complex memory trade-offs of shared-array fusion. The integration of spatial pipeline parallelism represents a separate architectural paradigm and remains a potential for future work.

3.1.3. Hardware Implementations of Layer Fusion

The theoretical benefits of layer fusion have been heavily substantiated by dedicated silicon and FPGA implementations, which demonstrate the empirical value of avoiding intermediate memory round-trips. For instance, classic fused-layer CNN accelerators have shown that layer fusion of initial five convolutional layers of the VGG-E (a weight dominant workload) layer fusion reduces off-chip data transfers by approximately 95% (from 77 MB down to 3.6 MB) while requiring only a modest increase in on-chip storage [2].

ISAAC and DepFiN Building upon these principles, the ISAAC [26] architecture exemplifies a pipelined, fused organization tailored for in-situ analog processing. Central to ISAAC's design is the memristor crossbar array, a dense hardware matrix where computational weights are stored as programmable resistances at the cross-points of conductive wires. This structure allows MAC operations to be performed directly in the analog domain as voltages pass through the grid, meaning the weights remain completely stationary. To enable fusion, eDRAM buffers stage inter-layer data, allowing subsequent layers to commence computation as soon as a minimal activation window is generated.

More recently, depth-first silicon implementations such as *DepFiN* [10] have pushed the boundaries of fusion for high-resolution pixel processing workloads. DepFiN relies on a control flow that supports frequent inter-layer switching to keep intermediate feature maps on-chip. This architecture achieves a reported on the MC-CNN-fast workload up to 18× system-level efficiency gains when compared to a conventional layer-by-layer processor

with equivalent core-level efficiency.

The tremendous physical performance gains demonstrated by hardware like ISAAC and DepFiN strongly motivate the need for generalized exploration frameworks, such as the one proposed in this thesis, capable of automatically navigating the complex scheduling and mapping spaces these accelerators introduce.

3.2. Degrees of Freedom in Fused Mapping

Section 2.4.1 identified three degrees of freedom for single-layer mapping: tiling, loop ordering, and parallelism strategy. All three remain relevant under fusion, but they are augmented by decisions that have no single-layer counterpart. This section organizes the fused-layer mapping degrees of freedom into three groups: *inter-layer* tiling choices (new), *intra-layer* decisions (inherited), and *memory-level bypass rules* (new).

3.2.1. Inter-Layer Tiling

Inter-layer tiling refers to the choice of *which dimensions are tiled at the outermost (DRAM) level* and *what tile sizes are assigned* to those dimensions. In a fused mapping, these decisions determine how the spatial extent of the computation is partitioned into tiles that cascade through the layers, as described in Section 3.1.2.

Three classes of dimension can be tiled at the inter-layer level:

Spatial (output) dimension tiling. The output spatial dimensions are the primary candidates for inter-layer tiling. This form of tiling is applicable regardless of the number of fused layers and is the most common inter-layer tiling strategy.

For example, in the DepFiN configuration validated in Chapter 6, the 1280-pixel output width is tiled into 10 tiles of 128 pixels each ($Q = 10$), and all 720 rows are iterated ($P = 720$).

Channel tiling. In a two-layer fusion, it is possible to tile the output channels of the first layer (equivalently, the input channels of the second) across multiple passes. However, for fusions of three or more layers this form of tiling is not applicable to internal intermediate channels with depth-first execution model, because it requires that each intermediate tile be fully produced and consumed within a single pass, while in this case I would have only a partial intermediate output tile after a pass.

Tile iteration order. Beyond choosing which dimensions to tile and their sizes, the mapper must decide the *iteration order* of the tiles: the sequence in which spatial tiles are visited. For a 2-D tiling with factors P and Q, the two orderings $\langle P, Q \rangle$ (row-major: complete one row of tiles before moving to the next) and $\langle Q, P \rangle$ (column-major: complete one column before moving to the next row) produce different data reuse patterns. In a row-major scan, horizontally adjacent tiles share vertical halo regions; in a column-major scan, vertically adjacent tiles share horizontal halo. The optimal choice depends on the filter shapes, strides and the relative costs of fetching horizontal vs. vertical non-overlapped slices.

3.2.2. Intra-Layer Decisions

Within each fused layer, the single-layer mapping (intra-layer mapping) degrees of freedom apply unchanged:

- **Tiling** at on-chip memory levels determines how each layer's tile (whose size is fixed by the inter-layer tiling and the cascading halo) is further subdivided for processing by the PE array (Section 2.4.1).
- **Loop ordering** at each memory level determines the dataflow, which operand is stationary and whether halo reuse is exploited (Section 2.4.1).
- **Spatial parallelism** at fanout levels determines how the PE array is used: which dimensions are unrolled across rows and columns (Section 2.4.1).

These three decisions are made independently for each layer in the fused group, but they all operate *within* the tile boundaries established by the inter-layer tiling. Changing the inter-layer tile size therefore ripples into every layer's intra-layer decisions: a smaller inter-layer tile means smaller per-layer tiles, potentially reducing intra-layer reuse.

3.2.3. Memory-Level Bypass Rules

In a single-layer mapping, three operand types (input, weight, output) must each be assigned to one or more memory levels, and levels that do not store a particular operand *bypass* it. Bypass decisions in single-layer mapping are relatively straightforward: a scratchpad either stores an operand or it does not, and the choice is driven by the level's capacity and bandwidth.

Fusion introduces five operand categories (Section 3.1.2), and the bypass configuration at each level must be specified for *each* of them. This creates a richer, and more consequen-

tial, design space. The bypass rules collectively determine where intermediate activations reside, and therefore whether the depth-first fusion guarantee is enforced. In fused-layer execution under depth-first scheduling, the memory bypass configuration is typically enforced as a rigid *per-architecture* design decision rather than a flexible per-mapping choice. Because these bypass rules are strictly fixed within the hardware description, they cannot be actively explored by the mapper as a degree of freedom. Instead, from the perspective of the analytical engine, these predefined bypass configurations act as fundamental hardware constraints that rigorously prune the search space by dictating which intermediate operand allocations are physically valid.

3.3. Data Reuse under Fusion

Section 2.3.3 has introduced the fundamental reuse mechanisms available in single-layer mapping: stationarity (temporal reuse), halo overlap, and spatial multicast. Section 2.7.3 then has noted that fusion introduces an additional form of reuse, inter-layer reuse, arising from the on-chip residency of intermediate activations.

This section provides a more detailed analysis of the reuse opportunities specific to fused execution, distinguishing between *immediate* and *buffered* reuse, and formalizing the *reuse–recompute trade–off* that governs the relationship between on-chip memory capacity and data-movement efficiency.

3.3.1. Immediate Reuse

Immediate reuse occurs when the intermediate input region required to compute a given output tile *overlaps* the intermediate input region used for the *immediately preceding* output tile in the schedule. Because the two tiles are processed consecutively, the overlapping elements are still resident in the on-chip buffer from the previous iteration; no additional storage beyond the current working set is needed.

This form of reuse arises naturally from the *halo* of convolutional filters. Consider a 1-Dimensional example with a 3×3 filter ($R = 3$, stride 1). The overlap between tile k and tile $k+1$ is $R-1 = 2$ elements. These elements were fetched (or produced by a preceding layer) during tile k and can be reused for tile $k+1$ without any re-fetch from DRAM, provided the iteration order visits tiles in spatial sequence.

Immediate reuse is cost-free in terms of memory: it requires no extra buffer capacity, only that the scheduling policy visits adjacent tiles consecutively. It is the halo-reuse mechanism described in Section 2.4.1, applied to the *inter-layer* context: the overlapping

data is the intermediate activation produced by a preceding layer rather than an input fetched from DRAM.

3.3.2. Buffered Reuse

Buffered reuse arises when the overlapping region between two tiles that are not consecutive but at a distance spans of multiple tiles.

A common scenario is 2-Dimensional tiling with a row-major iteration order. After completing all tiles in one row, the schedule advances to the next row. The first tile of the new row overlaps vertically with the first tile of the previous row by $R - 1$ rows of the intermediate feature map. However, all tiles of the previous row have been processed in between, so the overlapping data is no longer in the immediate working set.

To exploit this reuse, the accelerator must *retain* the overlapping region in the on-chip scratchpad for the duration of the entire row of tiles. The required residency, and therefore the extra scratchpad capacity needed, grows with the number of tiles per row and the overlap size:

$$S_{\text{buffered}} = (R - 1) \cdot W_{\text{row}} \cdot C, \tag{3.1}$$

where W_{row} is the full row width of the intermediate feature map and C the number of channels. This capacity must be reserved *in addition to* the working set of the current tile.

Buffered reuse trades scratchpad capacity for reduced data movement: the S_{buffered} elements are fetched (or produced) once and reused across multiple tile iterations, avoiding redundant re-computation or re-fetching from DRAM.

3.3.3. Reuse–Recompute Trade-off

The decision to buffer or to recompute is central to fused-layer mapping. Let S_{required} denote the total scratchpad capacity needed to realize a given level of buffered reuse (working set plus buffered overlap), and let $S_{\text{available}}$ denote the physically available on-chip capacity.

- If $S_{\text{required}} \leq S_{\text{available}}$, buffered reuse is feasible: the overlapping data is retained on-chip and reused, eliminating redundant computation and data movement.
- If $S_{\text{required}} > S_{\text{available}}$, the overlapping data must be *evicted* and either *re-fetched* from DRAM (if intermediate fetching from DRAM is allowed) or *recomputed* by re-executing the producing layer for the overlapping region.

The trade-off has three axes:

1. **Reuse reduces data movement.** Greater reuse eliminates redundant reads from outer memory levels, reducing both energy (fewer high-cost accesses) and latency (fewer bandwidth-limited transfers).
2. **Buffered Reuse increases scratchpad demand.** Retaining overlap regions consumes buffer capacity that could otherwise be used for larger tiles or for storing other operands (e.g., weights).
3. **Recomputation increases compute cost but relaxes storage.** Re-executing a producing layer for the halo region adds multiply accumulation (MAC) operations but avoids the need to buffer the overlap. In compute-bound regimes (where the PE array is fully utilized and memory bandwidth is not the bottleneck), the additional computation may be nearly free; in bandwidth-bound regimes, it adds non-negligible latency.

Design-space exploration must therefore sweep output-tile dimensions and scheduling policies and, for each candidate mapping, check the feasibility condition $S_{\text{required}} \leq S_{\text{available}}$ to determine whether buffered reuse is achievable. The optimal mapping balances reuse, recomputation, and on-chip constraints to minimize the target cost metric (energy, latency, or EDP).

3.3.4. The Need for Per-Layer Mapping and Flexible Execution Granularity

A naive strategy for transitioning from single-layer execution mapping a to multi-layer fusion mapping representation might attempt to mathematically compose the layers by completely merging their loop indices into a single, monolithic mathematical formula. However, this over-simplification mathematically eliminates the explicit representation of the intermediate feature maps. The intermediate spatial dimensions (e.g., intermediate height and width) lose their standalone loop variables, becoming inextricably bound to the shared outer dimensions of the final output and the overlapping filter halo.

Failing to maintain distinct, independent iteration variables for each layer introduces severe architectural modeling limitations, primarily manifesting in two ways:

1. **Loss of Per-Layer Mapping Freedom.** DNN architectures frequently compose layers with vastly different topologies (with respect to activation and weight size). Consequently, the optimal mapping will naturally differ for each layer. If a fused schedule forces all layers to share the exact same loop variables universally, the

mapping engine loses the ability to express heterogeneous dataflows. Every loop transformation applied to the first layer would rigidly and unavoidably transform the other layers. In a properly defined design space, the mapper must retain the freedom to assign distinct tilings and local dataflows (e.g., enforcing Weight-Stationary for Layer 0 and Output-Stationary for Layer 1) despite they are fused.

2. **Forced Scalar Lockstep (Back-to-Back MAC Execution).** If intermediate iteration variables are completely collapsed, the scheduling granularity is forced into an inflexible extreme. Because no loop variable can advance the computation of the producer without simultaneously advancing the consumer, the operations become locked into a strict *back-to-back* scalar schedule.

   In this restrictive model, at every innermost iteration, the producer executes a single Multiply-Accumulate (MAC) operation to generate one intermediate scalar, which the consumer must immediately fetch and compute. This eliminates any opportunity for temporal buffering, assuming that all the MACs operations happens at the same time.

Therefore, a robust formulation of the fused-layer design space must mathematically decouple the iteration spaces of the fused layers. It must explicitly represent intermediate dimensions to allow for distinct, per layer mapping choices.

3.3.5. The Need for Constraint-Based Pruning

The astronomical size of the fused map-space (Table 5.1) makes any form of exhaustive enumeration or even large-scale random sampling entirely infeasible. Unlike single-layer mapping, where random or heuristic search can still cover a meaningful fraction of the space within reasonable time, the fused map-space is so large that unguided search has effectively zero probability of discovering high-quality mappings.

The key insight that reduce this immense map-space is that a vast majority of the 10^{515} configurations are either *invalid* (they violate memory capacity constraints) or *structurally sub-optimal* (they contradict known properties of the hardware). **Constraint-based pruning** eliminates part of these configurations *before* evaluation by fixing dimensions and factors that are determined by the architecture:

1. **Dataflow constraints** fix the loop ordering at levels where the hardware enforces a specific dataflow. For example, in DepFiN all spatial dimensions are constrained to iterate at the DRAM level, and all weight dimensions are constrained to the Weight

Memory (WMEM) level. Each fixed ordering eliminates $|D|! - 1$ permutations at that level.

2. **Factor constraints** fix the tile sizes for dimensions whose factorization is dictated by the architecture. For instance, constraining $Q = 10$ at the DRAM level and $Z_i = 16$ at the spatial row level leaves only one valid factor allocation for those dimensions, eliminating an exponential number of invalid candidates.

3. **Bypass constraints** exclude operand–level combinations that are physically impossible (e.g., intermediate operands at DRAM), reducing the number of memory-level interactions to evaluate.

The critical point of constraint-based pruning is neither an approximation nor a heuristic that might discard high-quality solutions: the constraints encode *properties of the hardware* that any valid mapping must satisfy. The pruned space therefore contains all valid and potentially optimal mappings, no feasible solution is lost.

4 | Related Works

This chapter reviews the literature and technological landscape that contextualizes the proposed dataflow-level framework for scheduling multiple fused layers.

4.1. Exploring Layer-By-Layer Dataflows

With the advent of spatial architectures (SAs), the *mapping problem* became a central concern. This, in turn, created a need for general techniques applicable across diverse SAs and workloads. Research has progressed along two complementary fronts: (i) analytical modeling frameworks that capture the salient microarchitectural features of SAs to provide fast, reasonably accurate performance and energy estimates for candidate mappings, and (ii) automated mappers that use these models as feedback to explore the map-space and identify high-quality schedules and dataflows [21, 22].

4.1.1. Timeloop

Timeloop [21] is a foundational infrastructure for evaluating and exploring the architectural design space of deep-learning accelerators based on SAs. Its objective is to place different accelerators on a level playing field with respect to mappings, thereby providing insights that can guide future SA designs. To that end, it adopts a mapper–model paradigm: the mapper searches for high-quality mappings that the model can use to fairly compare SAs, while the model supplies feedback to the mapper during the search. This innovation has made Timeloop a reference point for subsequent work in the area.

At the core of its analytical model, Timeloop introduces a nested-loop representation of workloads, primarily convolutions, in which each loop-body iteration corresponds to a Multiply-and-Accumulate (MAC) operation. Unfortunately, Timeloop is not efficient for simpler but very common workloads, like GEneral Matrix Multiplications (GEMMs). Mappings are then expressed as tilings and permutations of these loops across the SA's memory hierarchy and PE mesh; however, spatial fanout levels are handled implicitly. From these abstractions, Timeloop constructs the map-space for any SA–convolution

pair, focusing particularly on the unconstrained map-space.

Given this representation, computation and data-movement patterns in neural-network kernels can be inferred analytically, enabling Timeloop to estimate throughput and access counts. By combining these results with Accelergy [30], Timeloop achieves rapid estimations, avoiding the need for slow gate-level simulations after physical layout, while maintaining 95% accuracy against well-known SAs like Eyeriss. Because Timeloop was designed primarily as a comparative tool for SAs, its map-space exploration strategies are intentionally simple, relying on exhaustive or random search to ensure consistency within modest runtimes.

4.1.2. ZigZag

ZigZag [19] is an analytical framework and map-space exploration engine designed to evaluate and compare spatial architectures. While tools like Timeloop maintain a strict conceptual separation between tiling and loop ordering, ZigZag introduces an enhanced nested-loop abstraction that unifies these decisions. In this model, every prime factor of a workload dimension is assigned its own distinct loop. This allows dataflows, temporal reuse, and spatial fanouts to be determined simultaneously by binding specific loops and operands directly to memory levels.

To evaluate hardware utilization, stall cycles, and energy consumption, ZigZag relies on the principle of "loop relevance." This concept is mathematically equivalent to the orthogonality and stationarity principles utilized in our framework, where a loop iterating over a dimension orthogonal to an operand effectively reuses that operand, rendering outer memory accesses irrelevant for that cycle.

For map-space exploration, the original ZigZag framework proposed a multi-stage mapper heavily reliant on aggressive pruning heuristics. By aggressively filtering out mappings based on spatial utilization thresholds and suboptimal data stationarity, ZigZag could reduce the search space by orders of magnitude. However, this heavy reliance on pruning inherently risks eliminating optimal solutions, resulting in an estimated performance loss compared to an exhaustive search.

4.1.3. FactorFlow

FactorFlow [22] is a highly efficient analytical modeling and mapping framework tailored specifically for General Matrix Multiplications (GEMMs) and convolutions. The architectural model is fully flexible and expressed via configurable components that strictly

reflect the hierarchical memory abstraction detailed in Section 2.6.

Compared to Timeloop, FactorFlow provides explicit and highly detailed modeling of spatial fanout levels, capturing the intricate physical interconnections between inner processing instances and the broader memory hierarchy. It introduces precise architectural toggles for multicast and reduction capabilities, defining specific on-chip communication topologies (e.g., forwarding, one-to-one, one-to-many). Furthermore, every architectural level can independently specify its supported dataflows; the creation of the concept of spatial fanouts, in particular, explicitly constrain which dimensions can leverage reuse when spatially unrolled.

The primary motivation for selecting FactorFlow as the foundational baseline for this thesis lies in exceptional computational efficiency, a critical requirement heavily emphasized by the industry-wide architectural shift toward Transformer networks, where GEMMs represent the dominant computational bottleneck. While other state-of-the-art exploration tools technically support GEMMs, they fundamentally treat them as degenerate convolutions (with three of the six spatial dimensions artificially restricted to one). This generalized abstraction introduces severe analytical overheads and relies on mapping heuristics that fail to exploit GEMM-specific mathematical characteristics, causing these tools to struggle when searching for optimal mappings within a reasonable timeframe [22].

FactorFlow mathematically circumvents these limitations, yielding an analytical model that is significantly more lightweight. It consistently discovers near-optimal mappings, achieving a sensitive improvement in Energy-Delay Product (EDP) and up to 205× faster execution times compared to alternative state-of-the-art frameworks [22]. Furthermore, by integrating the QuickFlow local search method [23], FactorFlow effectively inverts the traditional mapping paradigm. By prioritizing the search for optimal factor allocations (tiling and parallelism) and subsequently resolving the optimal loop permutation in a single shot, the framework achieves up to 182× faster runtimes even for standard convolutions.

For a thesis aiming to formalize and navigate the fused-layer design space, where the mapspace expands exponentially with each additional fused layer, relying on an evaluator that drastically reduces baseline analytical overhead is not merely a software optimization; it is a strict mathematical and computational necessity.

4.2. Fused-Layer DF Exploration Framework

4.2.1. DeFiNES

DeFiNES [20] introduces a comprehensive framework for exploring the depth-first (DF) scheduling space. It builds upon ZigZag (see Section 4.1.2, however the evaluation of mappings lacks the time efficiency demonstrated by more recent works [22], which can limit the scalability of the exploration for a mapper. DeFiNES formalizes the depth-first design space along three primary axes: (i) the tile size at the final layer, (ii) the overlap storing mode (i.e., caching policies), and (iii) the fuse depth.

A key contribution of DeFiNES is its unified analytical cost model for estimating latency and energy. Unlike prior works that often limit their scope to DRAM and activation memory accesses, DeFiNES accounts for the full on-chip multi-level memory hierarchy and explicitly models weight traffic. By evaluating the system comprehensively, DeFiNES demonstrates that partial architectural models can be misleading, reporting up to a 10× improvement in the quality of found solutions when both full memory hierarchy and weight costs are considered. The approach used in DeFiNES introduces an analytical wrapper around the standard single-layer ZigZag optimizer. Unfortunately, DeFiNES restricts cross-layer tiling strictly to spatial dimensions (horizontal and vertical), explicitly prohibiting cross-layer tiling across channel dimensions. As highlighted by recent literature [9], restricting tiling choices limits overall tensor reuse; the diverse shapes of DNN layers dictate that no single spatial tiling strategy is universally optimal. By introducing the capability to tile across channel dimensions, the scheduling approach proposed in this thesis explores a **richer design space, capturing inter-layer reuse opportunities that spatial-only frameworks inherently miss.**

4.2.2. LoopTree

LoopTree [9] proposes an extensive design space and a corresponding analytical model to evaluate latency, energy, buffer capacity, and off-chip transfers for fused-layer dataflows. It models complex trade-offs involving inter-layer tiling, data retention, and recomputation.

While LoopTree excels as a highly accurate analytical model and baseline comparator, its primary weakness lies in its map-space exploration mechanism. Because LoopTree is built as an extension of Timeloop, it inherits Timeloop's generalized, exhaustive mapping strategies. Within the context of layer-by-layer optimization, exhaustive search is merely slow; however, in a fused-layer paradigm, where the map-space grows exponentially with each additional fused layer and tile size variation, this approach struggles

to scale. Consequently, while LoopTree serves as a robust analytical comparator, it is severely bottlenecked during the search process, making it difficult to rely on for finding optimal solutions **within a competitive runtime**.

A further distinction between LoopTree and the framework proposed in this thesis lies in the representational approach. LoopTree **introduces a tree-based taxonomy to specify inter-layer dependencies and dataflows, which represents a structural departure from traditional nested-loop abstractions**. **In contrast, our approach proposes a cohesive loop nest abstraction** and organic extension to the existing FactorFlow syntax, maintaining structural consistency while seamlessly introducing fused-layer capabilities.

4.3. Broader Landscape of Layer Fusion

Beyond the foundational frameworks and hardware implementations discussed above, the state-of-the-art literature explores specific facets of the layer fusion paradigm. Works such as *ConvFusion* [29], early fused-layer CNN architectures [2], and hardware-software co-design strategies like HW Flow Fusion [28] have proposed targeted techniques to minimize intermediate data transfers. While these contributions provide valuable optimizations for specific network topologies or fixed hardware dataflows, they generally lack the generalized, search-based mapping capabilities necessary to automatically optimize schedules across diverse spatial architectures.

When discussing automated fusion exploration, it is critical to distinguish between macro-level graph decisions and micro-level dataflow mapping. *Optimus* [5] exemplifies a state-of-the-art approach to the former. Operating as a graph-level compiler, Optimus employs memory-aware heuristics to mathematically determine exactly *which* layers within a neural network should be grouped into fusion segments to avoid memory bottlenecks.

Crucially, graph-level compilers like Optimus are strictly *complementary* to the analytical mapping framework developed in this thesis. While Optimus solves the topological problem of selecting the optimal fusion sets, it abstracts away the complex, low-level hardware scheduling. Conversely, our framework solves the spatial mapping problem, determining exactly *how* to tile, order, and physically execute those fused layers on a specific hardware accelerator to maximize reuse and efficiency. Ultimately, pairing a graph-level topology scheduler like Optimus with the fine-grained, constraint-aware dataflow engine proposed in this work represents a clear path toward a fully autonomous, end-to-end compilation pipeline for deep learning accelerators.

5 | Proposed Solution

Chapter 3 has characterized the fused-layer mapping design space: the proliferation of dimensions (Section 3.1.2), the five operand categories (Section 3.1.2), the cascading tile dependencies (Section 3.1.2), and the combinatorial growth of the map space (Section 5.1.3). This chapter presents how we have **extended FactorFlow** to model, evaluate, and navigate that space. The extension rests on three conceptual pillars, presented in Section 5.1: *aliasing*, which captures the two-sided view of every intermediate activation, produced by one layer, consumed by the next; *decoupling*, which guarantees each layer's tiling remains locally independent; and a *producer–consumer feasibility pruning* rule, which eliminates infeasible mappings by enforcing the producer–consumer tile-size inequality at every hierarchy level.

Section 5.2 describes the concrete data structures that extend FactorFlow's workload, architecture, and mapping specifications to multi-layer fusion. Section 5.4 develops the per-layer cost-model extensions, memory operations, energy, and latency with bandwidth-limited stall detection, that generalize the single-layer model of Section 2.6. Section 5.3 discusses the implementation: the overall code structure and the current limitations.

5.1. Modeling Fused Computation

In our work, a fused group of F consecutive convolution layers executes as a single, unified loop nest over a combined set of dimensions $\mathcal{D}$. Three modeling concepts govern how we represents the inter-layer relationships within this unified nest.

5.1.1. Dimension Aliasing

When layers are fused, each intermediate activation A_i $(0 \leq i \leq F-2)$ serves a dual role: it is the *output* of layer i and the *input* of layer $i+1$. The central modeling question is: which loop variables should index this shared tensor?

The Problem: Naming Intermediate Dimensions

In a single-layer convolution the output is indexed by its own native dimensions (Z, P, Q) and the input is indexed through affine dim-sums $([P, R], [Q, S])$, as discussed in Section 2.2.3. When two convolutions are fused the intermediate tensor connects them: the output dimensions of the first convolution become the input dimensions of the second.

Expressing the input of the second convolution in terms of the final output would require *nested dim-sums*. For a two-layer fusion ($F = 2$) where the second convolution reads at position $(p + r_1,\ q + s_1)$ and the first convolution must produce tiles large enough to cover that range, the first convolution's input would be indexed as:

$$\mathrm{In}[\, c_0,\ \underbrace{(p + r_1)}_{\text{int. height}} + r_0,\ \underbrace{(q + s_1)}_{\text{int. width}} + s_0\,] \ \equiv\ \mathrm{In}[\, c_0,\ [\,[P, R_1],\ R_0\,],\ [\,[Q, S_1],\ S_0\,]\,].$$

For an F-layer fusion this nesting would grow to depth $F-1$, producing constructs such as $[[\cdots[[P, R_{F-1}], R_{F-2}], \cdots], R_0]$.

Nested dim-sums make it impossible to iterate over the intermediate height and width *independently* per layer: those dimensions have no names of their own and therefore cannot appear as standalone loops in the mapping. This has two severe consequences:

1. **No per-layer mapping freedom.** Without independent intermediate dimensions the mapping cannot express a different tiling or dataflow for each layer. Every loop that moves the first convolution simultaneously moves the second, because the only available variables (P, Q, R_1, S_1, R_0, S_0) are shared across the nested dim-sum, the fused layers cannot have independent stationarities (Section 2.3.3).

2. **Forced back-to-back multiply accumulation (MAC) execution.** Because no loop variable can advance the first convolution without simultaneously advancing the second, the two MAC operations are locked into a *back-to-back* schedule: at every innermost iteration, MAC_0 produces one intermediate element and MAC_1 immediately consumes it. The mapping has no mechanism to express, for instance, performing several iterations of MAC_0 (accumulating a partial tile of the intermediate) before switching to MAC_1 to consume that tile, even though such a schedule could improve weight reuse or reduce partial-sum traffic. More broadly, neither a layer-by-layer schedule (complete all MAC_0 iterations before any MAC_1) nor any intermediate granularity is representable; the only schedule is strictly element-wise interleaving.

The Solution: Concrete Intermediate Dimensions

Aliasing resolves this by introducing a *new set of loop variables* for each intermediate tensor. The idea is to give the intermediate its own concrete dimension names, for instance Y_0 and X_0 for the height and width of the first intermediate activation, and to register them in the coupling (Section 2.2.2) so that:

- the **producer** (layer 0) indexes the intermediate using these names as *native* dimensions, just as a single-layer convolution indexes its output;
- the **consumer** (layer 1) indexes the *same* tensor through a *dim-sum* involving its own variables, treating the intermediate, and the physical tensor, as its input.

Formally, for the connection between layer i and layer $i+1$ and the tensor A_i, we defines two coupling entries:

- A **producer coupling** (`int_out`$[i]$) that indexes A_i with native dimensions:

$$A_i[\, z_i,\ y_i,\ x_i\,].$$

- A **consumer coupling** (`int_in`$[i]$) that indexes the same tensor through affine dim-sums:

$$A_i[\, c_{i+1},\ y_{i+1}{+}r_{i+1},\ x_{i+1}{+}s_{i+1}\,].$$

The dimensions Y_i and X_i are now concrete: they appear in the unified loop nest, can be tiled, reordered, and parallelized just like any other dimension. At the same time, the consumer sees the intermediate through its own names (Y_{i+1}, R_{i+1}), so that each layer's coupling remains a one-level list, no nesting required.

Two-Layer Convolution Example

Consider the simplest non-trivial case: a fusion of two 2-D convolutions ($F = 2$). The complete coupling, as defined in the implementation, is:

Operand	Coupling entry	Indexing
`in`	$[\,C_0,\ [Y_0, R_0],\ [X_0, S_0]\,]$	$\mathrm{In}[\,c_0,\ y_0{+}r_0,\ x_0{+}s_0\,]$
`w`[0]	$[\,Z_0,\ C_0,\ R_0,\ S_0\,]$	$W_0[\,z_0,\ c_0,\ r_0,\ s_0\,]$
`int_out`[0]	$[\,Z_0,\ Y_0,\ X_0\,]$	$A_0[\,z_0,\ y_0,\ x_0\,]$
`int_in`[0]	$[\,C_1,\ [P, R_1],\ [Q, S_1]\,]$	$A_0[\,c_1,\ p{+}r_1,\ q{+}s_1\,]$
`w`[1]	$[\,Z_1,\ C_1,\ R_1,\ S_1\,]$	$W_1[\,z_1,\ c_1,\ r_1,\ s_1\,]$
`out`	$[\,Z_1,\ P,\ Q\,]$	$\mathrm{Out}[\,z_1,\ p,\ q\,]$

The twelve unified dimensions are $\{C_0,\ Y_0,\ X_0,\ R_0,\ S_0,\ Z_0,\ C_1,\ R_1,\ S_1,\ Z_1,\ P,\ Q\}$, and the two MAC operations described by this coupling are:

$$\mathrm{MAC}_0: \quad A_0[\,z_0,\ y_0,\ x_0\,] \mathrel{+}= W_0[\,z_0,\ c_0,\ r_0,\ s_0\,] \cdot \mathrm{In}[\,c_0,\ y_0{+}r_0,\ x_0{+}s_0\,], \tag{5.1}$$

$$\mathrm{MAC}_1: \quad \mathrm{Out}[\,z_1,\ p,\ q\,] \mathrel{+}= W_1[\,z_1,\ c_1,\ r_1,\ s_1\,] \cdot A_0[\,c_1,\ p{+}r_1,\ q{+}s_1\,]. \tag{5.2}$$

The key property is visible in the coupling table: the rows for `int_out`[0] and `int_in`[0] index the *same* tensor A_0 under *different* names. Each pair of dimensions that addresses the same axis of an intermediate tensor constitutes an *alias*:

- **Channel alias.** Z_0 (producer: number of output filters) $\leftrightarrow$ C_1 (consumer: number of input channels). Both index the channel axis of A_0.
- **Spatial aliases.** Y_0 (producer: output height) $\leftrightarrow$ $[P, R_1]$ (consumer: affine dim-sum addressing input height); X_0 (producer: output width) $\leftrightarrow$ $[Q, S_1]$ (consumer: affine dim-sum addressing input width).

In a general F-layer fusion each intermediate connection i ($0 \le i \le F-2$) introduces the same three aliases, with the last connection using the global output variables P, Q and the first connection using the global input coupling.

Aliasing is *implicit* in the data structures: no explicit pointer links Z_i to C_{i+1} or Y_i to $Y_{i+1} + R_{i+1} - 1$. Instead, the feasibility constraints are enforced by the producer–consumer feasibility pruning (Section 5.1.4), which is evaluated after every candidate mapping modification.

5.1.2. Dimension Decoupling

Aliasing gives each intermediate tensor concrete dimension names that appear in the unified loop nest. A fundamental consequence of this naming is that the dimensions of different layers become disjoint, leading to a property we call *decoupling*: iterating over any dimension of one layer leaves all other layers *stationary*.

Per-Layer Dimension Sets

The unified dimension set $\mathcal{D}$ can be partitioned into per-layer subsets by collecting, for each layer, every dimension that appears in any operand coupling involving that layer. For the two-layer example of Equations 5.1–5.2:

$$\mathcal{D}_0 = \underbrace{\{C_0, Y_0, R_0, X_0, S_0\}}_{\texttt{in}} \cup \underbrace{\{Z_0, C_0, R_0, S_0\}}_{\texttt{w}[0]} \cup \underbrace{\{Z_0, Y_0, X_0\}}_{\texttt{int\_out}[0]} = \{C_0, Y_0, X_0, R_0, S_0, Z_0\}, \tag{5.3}$$

$$\mathcal{D}_1 = \underbrace{\{C_1, P, R_1, Q, S_1\}}_{\texttt{int\_in}[0]} \cup \underbrace{\{Z_1, C_1, R_1, S_1\}}_{\texttt{w}[1]} \cup \underbrace{\{Z_1, P, Q\}}_{\texttt{out}} = \{C_1, P, Q, R_1, S_1, Z_1\}. \tag{5.4}$$

Crucially, $\mathcal{D}_0 \cap \mathcal{D}_1 = \emptyset$: the two sets share *no* dimension name. This disjointness is precisely what aliasing achieves, Z_0 and C_1 refer to the same physical channel axis of A_0, but they are syntactically distinct loop variables, each belonging to exactly one layer's set.

In the general F-layer case, an intermediate layer i $(0 < i < F-1)$ collects its dimensions from the weight coupling, the preceding `int_in`, and the succeeding `int_out`:

$$\mathcal{D}_i = \underbrace{\{C_i, Y_i, R_i, X_i, S_i\}}_{\texttt{int\_in}[i-1]} \cup \underbrace{\{Z_i, C_i, R_i, S_i\}}_{\texttt{w}[i]} \cup \underbrace{\{Z_i, Y_i, X_i\}}_{\texttt{int\_out}[i]}.$$

Each $\mathcal{D}_i$ is disjoint from every $\mathcal{D}_j$ $(j \neq i)$ because the subscript index supplied by aliasing ensures that no dimension name is reused across layers.

Stationarity and Independent MAC Counting

At the innermost point of the unified loop nest, all operands are indexed down to a single element. In a two layer fusion, two possible MAC operations exist, one per layer, among these elements:

$$\text{MAC}_0 : \; A_0 \mathrel{+}= W_0 \cdot \text{In}, \qquad \text{MAC}_1 : \; \text{Out} \mathrel{+}= W_1 \cdot A_0.$$

Because $\mathcal{D}_0$ and $\mathcal{D}_1$ are disjoint, advancing any dimension in $\mathcal{D}_0$ changes the indices of MAC_0's operands while leaving every index of MAC_1 *fixed*. From the perspective of MAC_1, nothing has moved; the second convolution is *stationary*. Symmetrically, iterating over any dimension in $\mathcal{D}_1$ leaves MAC_0 stationary.

Example for a 2 layer fusion. Consider iterating over X_0 (the intermediate width, a native dimension for layer 0). Each step of the X_0 loop advances the write position $A_0[\ldots, x_0]$ and the read position $\text{In}[\ldots, x_0 + s_0]$; new MAC_0 operations are performed. Meanwhile, $X_0 \notin \mathcal{D}_1$: neither A_0, W_1 as read by the consumer (which uses the dim-sum $[Q, S_1]$, not X_0), nor Out change their indices, so no MAC_1 operation is triggered.

Conversely, iterating over Q (the output width, a layer-1 dimension) advances the consumer's read position $A_0[\ldots, q+s_1]$ and the output write position $\text{Out}[\ldots, q]$; new MAC_1 operations are performed. Since $Q \notin \mathcal{D}_0$, the producer's indices remain fixed and no MAC_0 is executed.

Decoupling has two consequences for the mapper:

1. The dataflow, loop ordering, can be different for each layer
2. The tiling can be different for each layer

The Need for Pruning

Decoupling gives the mapper freedom to tile and reorder each layer's dimensions independently, but it introduces a risk: the unified loop nest no longer *automatically* guarantees that every intermediate value is computed before it is consumed.

Because A_0 is viewed from two different perspectives, written at position (z_0, y_0, x_0) by the producer and read at $(c_1, p+r_1, q+s_1)$ by the consumer, the mapper might choose a loop ordering or tiling that causes the consumer to request a position not yet written by the producer. A naïve non-sense mapping could, for instance, tile Z_0 more finely than C_1, forcing the consumer to expect more channels than the producer has completed.

This is the fundamental motivation for the producer–consumer feasibility pruning described in the section (Section 5.1.4): a constraint check that verifies, at every hierarchy level, that the producer's tile sizes are large enough to cover the consumer's access pattern, ensuring that the data dependencies imposed by aliasing are never violated.

5.1.3. Map-Space Growth under Fusion

Section 2.5 derived the map-space cardinality formula for a single-layer mapping (Equation 2.10). It is useful to apply this formula to the fused multi-layer configuration with aliasing and decoupling to quantify how the design space grows with the number of fused layers, and motivates the need of pruning and apply heuristics on the exploration space.

Applying the Cardinality Formula Recalling the map-space cardinality (Equation 2.10):

$$|\mathcal{MS}| = \underbrace{(|D|!)^{L_m}}_{\text{loop permutations}} \cdot \underbrace{\prod_{d \in D} \prod_{p|d} \binom{v_p(d) + L - 1}{L - 1}}_{\text{factor allocations}}.$$

Dimension Proliferation with Aliasing and Decoupling Aliasing and decoupling do not merely rename existing axes: they *creates new loop variables* that the mapper must tile and order. For each layer added to the fusion, the coupling introduces six fresh dimensions:

- Three **aliased dimensions** (Y_i, X_i, Z_i) that name the intermediate activation's height, width, and channel axes. Physically, these axes already exist, the intermediate tensor has a height, a width, and a depth that could be indexed simply with a nested dim-sum. However, without aliasing these axes have no independent names and cannot appear as standalone loops (see the nested dim-sums of Section 5.1.1). Once aliased, they become concrete dimensions visible to the mapper, each admitting its own tiling factors and its own position in the dataflow ordering at every level.
- Three genuinely new dimensions (C_i, R_i, S_i) that belong exclusively to the new layer's weight coupling. They introduce additional degrees of freedom for weight tiling and loop ordering (dataflow).

The total dimension count therefore grows by six for every fused layer.

The map-space impact is substantial. Each new dimension contributes additional permutations at every memory level (widening the dataflow search) and additional factor-allocation choices across levels (deepening the tiling search).

An example: Fused 10-layer configuration. For the 10-layer fusion workload validated in Chapter 6, the unified loop nest has $|D_{\text{fused}}| = 66$ dimensions, 6 new dimensions introduced for every layer fused. If, for instance, we want to execute this workload on

Eyeriss-like architecture: it has $L_m = 5$ memory levels (DRAM, Global Buffer, InputRegister, WeightRegister, IntermediateOutputRegister) and $L = 7$ total levels (5 memory + 2 spatial fanout levels).

The permutation term becomes:

$$(66!)^5 \tag{5.5}$$

Even a rough estimate shows the scale: $66! \approx 5.4 \times 10^{92}$, so $(66!)^5 \approx 10^{460}$, compared to $720^4 \approx 10^{11}$ for the single-layer case. The permutation term alone grows by over **400 orders of magnitude**.

The factor-allocation term also grows substantially. With 66 dimensions instead of 6, even if each dimension has only a single prime copy ($v_p(d) = 1$), the per-dimension allocation count is $\binom{1+7-1}{7-1} = 7$, and across 66 dimensions:

$$7^{66} \approx 10^{55}.$$

With larger extents (more prime copies), this term grows further.

Combined estimate. Combining both terms, the fused map-space exceeds $10^{460} \times 10^{55} = 10^{515}$ even under conservative assumptions.

Table 5.1 summarizes the comparison.

Table 5.1: Map-space size comparison: single layer vs. 10-layer fusion on a DepFiN-like architecture.

Quantity	**Single layer**	**10-layer fusion**
Dimensions ($\|D\|$)	6	66
Memory levels (L_m)	5	5
Total levels (L)	7	7
Permutation term	$\sim 10^{11}$	$\sim 10^{460}$
Factor-allocation term	$\sim 10^{13}$–10^{20}	$\sim 10^{55}$
Total map-space	$\sim 10^{24}$–10^{32}	$\sim 10^{515}$

5.1.4. Producer–Consumer Feasibility Pruning

Aliasing (Section 5.1.1) establishes that intermediate activations are shared between a producer and a consumer layer. This sharing imposes a hard physical constraint: the consumer cannot read a datum that the producer has not yet written. Any mapping that violates this constraint is *infeasible*, it would cause incorrect execution regardless of

the scheduling strategy (depth-first, pipelined, or otherwise). The following pruning rule encodes this constraint and is used to eliminate infeasible mappings from the search space *before* cost evaluation.

Cross-Level Validation

The core of the pruning logic is the observation that, in a multi-memory level loop nest, the producer dimension (i.e.: X, Y, Z) may be tiled across several hierarchy levels, while the consumer dimensions (i.e.: Q and S, P and R, C) that fall *between* two consecutive producer loop instances can also belong to different levels. For instance, considering the cross-level validation for the producer dimension width X. The constraint must therefore be checked not at each memory level in isolation, but at each boundary defined by the X loops in the nest, a boundary that can span multiple memory levels.

Zone definition. Consider a loop order that contains K instances of the producer dimension X: $x_1, x_2, \ldots, x_K$ (from outermost to innermost). Each consecutive pair (x_k, x_{k+1}) defines a *zone*, the set of all loops strictly between x_k and x_{k+1} in the nest. Within a zone, the consumer loops Q and S may appear at any memory level; the zone itself typically spans the boundary between two or more hierarchy levels.

Accumulated iterations. For each zone, the validator computes:

- Q_{zone}: the product of iterations of all Q loops in the zone (regardless of which memory level they belong to).
- S_{zone}: the product of iterations of all S loops in the zone.
- X_{cum}: the product of iterations of the current X loop and all X loops inner to it (i.e. the cumulative producer tile size from this boundary inward).

The constraint then requires:

$$Q_{\text{zone}} + S_{\text{zone}} - 1 \;\leq\; X_{\text{cum}}. \tag{5.6}$$

At the *outermost* X loop, the zone extends to the top of the nest, so Q_{zone} and S_{zone} become the *total global* iterations of Q and S across all levels, and X_{cum} becomes the total extent of X. This outermost check ensures that the full-dimension constraint $Q + S - 1 \leq X$ is satisfied globally.

Initial Condition

An additional constraint addresses the very bottom of the loop nest. The innermost loop with more than one iteration must be a *producer* (input spatial) dimension, not a consumer or weight dimension. This guarantees that the first non-trivial iterations of the nest produce intermediate data before any consumer iteration attempts to read it. If a consumer dimension Q or S has non-trivial iterations below the innermost X, the mapping is rejected immediately.

Exactness. This rule is not an approximation: it encodes a necessary physical property of any valid fused-layer schedule. No mapping that satisfies the constraints is discarded. The pruning is therefore *exact*: it removes only infeasible points from the map space, preserving every feasible, and potentially optimal, mapping.

Per-axis decomposition. Each intermediate layer i introduces three alias pairs (Section 5.1.1), one for each physical axis of the tensor A_i: width (Q_i, S_i, X_i), height (P_i, R_i, Y_i), and channels (C_{i+1}, Z_i). The constraint is analogous for each physical axis, with a simplification for the Z axis (because there is not an affine dimension in that case).

$$P_{\text{zone}} + R_{\text{zone}} - 1 \;\leq\; Y_{\text{cum}}. \tag{5.7}$$

$$C_{i+1,\text{zone}} \;\leq\; Z_{i,\text{cum}}. \tag{5.8}$$

Since the three axes index orthogonal dimensions of the same tensor, the pruning constraint can be evaluated *independently* for each triplet: satisfying the width constraint (Equation 5.6) neither implies nor precludes satisfaction of the height constraint (Equation 5.7), and likewise for the channel constraint (Equation 5.8). The implementation exploits this decomposition by calling the same single-triplet validator specifically for the axis that was moved by move-factor with respect to the starting valid configuration.

Two layer fusion example

To make the zone-based pruning concrete, this section walks through the width axis (Q, S, X) of a two-layer fusion (height and channel axes are validated independently with the same procedure). Throughout this example, subscripts on dimension names denote the *hierarchy level* (not the layer index): Q_1 is the Q loop at the DRAM level, X_3 is the X loop at the Register level, and so on.

Setup. Consider a width-axis configuration with full extents $X = 18$, $Q = 16$, $S = 3$ (satisfying the global constraint $Q + S - 1 = 18 = X$, with stride width = 1). The architecture has three memory levels: DRAM (level 1), Global Buffer (level 2), and Register (level 3). A candidate mapping tiles these dimensions across the three levels as follows (outermost to innermost):

Loop	Iterations	Level
Q_1	4	DRAM
X_1	3	DRAM
S_1	1	Global Buffer
Q_2	4	Global Buffer
X_2	2	Global Buffer
S_2	1	Register
Q_3	1	Register
S_3	3	Register
X_3	3	Register

The three X loops (X_1, X_2, X_3) define the zone boundaries. Between each pair of consecutive X loops, the validator collects the Q and S iterations that fall in that region, regardless of which memory level they belong to.

Initial condition. The innermost loop with more than one iteration is X_3 (3 iterations), which is a producer dimension. The initial condition is satisfied: the first non-trivial iterations of the nest produce intermediate data before any consumer loop attempts to read.

Zone-by-zone validation. The three X loops create three zones, validated from innermost outward. Table 5.2 summarizes the check.

Table 5.2: Zone-by-zone width-axis pruning validation for the example mapping. Every zone satisfies $Q_{\text{zone}} + S_{\text{zone}} - 1 \leq X_{\text{cum}}$.

Zone	Loops in zone	Q_{zone}	S_{zone}	X_{cum}	Check
1 ($X_3 \to X_2$)	S_2, Q_3, S_3	1	$1 \times 3 = 3$	3	$1+3-1 = 3 \leq 3$ ✓
2 ($X_2 \to X_1$)	S_1, Q_2	4	1	$2 \times 3 = 6$	$4+3-1 = 6 \leq 6$ ✓
3 (global)	Q_1	$4 \times 4 \times 1 = 16$	$1 \times 1 \times 3 = 3$	$3 \times 2 \times 3 = 18$	$16+3-1 = 18 \leq 18$ ✓

Since the check is valid for all the zones: the mapping is feasible on the width axis (as it is shown in Figure 5.1).

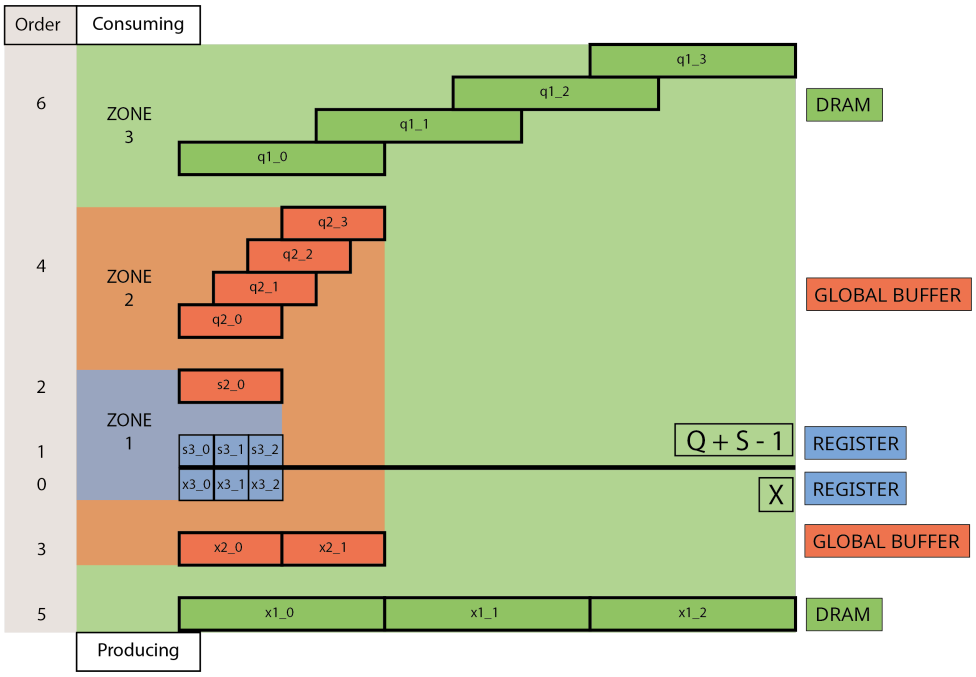

Figure 5.1: Graphic visualization of zone-by-zone width-axis pruning validation for the example mapping

A failing counterexample. To illustrate what the pruning catches, consider the *same* loop ordering but with a different tiling: move one factor of X from the Global Buffer level to the DRAM level, setting $X_1 = 6$ and $X_2 = 1$. Zone 2 now fails:

$$Q_{\text{zone}} + S_{\text{zone}} - 1 = 4 + 3 - 1 = 6 > 3 = X_{\text{cum}}.$$

The pruning validator rejects this mapping immediately, preventing the cost model from evaluating an execution schedule that would read data not yet produced.

Integration with the Mapper

Every candidate mapping move (Section 5.3.2) can trigger a call to the pruning validator. If the move results in a mapping that violates any of Equations 5.7, 5.8, the move is rolled back and marked as infeasible. Because the pruning eliminates only infeasible mappings, the mapper is guaranteed to retain every valid candidate.

5.2. Extending the FactorFlow Abstraction

FactorFlow describes a mapping problem through three inputs: a *workload*, an *architecture*, and a *mapping*. Each must be extended to support multi-layer fusion. The extensions

are designed to be *backward-compatible*: a single-layer problem is a special case with $F = 1$ and no intermediate operands.

5.2.1. Workload Specification

A workload is defined by two objects: a **Shape** and a **Coupling** 2.2.2.

The `Shape` is a dictionary mapping every dimension name to its size. For a fused group of F convolution layers operating on 2-D feature maps, the shape contains entries for:

- C_0: input channels; and, for the first layer, Y_0 and X_0 (the intermediate spatial extent produced by layer 0).
- For each layer $i = 0, \ldots, F-1$: Z_i, C_i, R_i, S_i (output channels, input channels, filter height, filter width).
- For each intermediate $i = 1, \ldots, F-2$: Y_i and X_i (spatial height and width of the intermediate activation produced by layer i).
- P, Q: final output height and width; Z_{F-1}: final output channels.

The `Coupling` encodes which dimensions index which operand (or tensor). Its constructor accepts *five* coupling lists, one for each operand category introduced in Section 3.1.2:

Argument	Type	Semantics
`in_coupling`	`list`	Dims indexing the global input
`w_coupling`	`dict[int → list]`	Per-layer weight dims
`out_coupling`	`list`	Dims indexing the global output
`int_out_coupling`	`dict[int → list]`	Per-connection producer dims
`int_in_coupling`	`dict[int → list]`	Per-connection consumer dims

The dictionary key for `w_coupling` is the layer index $i \in \{0, \ldots, F\text{–}1\}$; for the intermediate couplings it is the connection index $i \in \{0, \ldots, F\text{–}2\}$. Internally, the `Coupling` class stores both the raw (nested) and a flattened version of each coupling list. The flattened form is used for efficient look-ups during MOPs computation and constraint checking; the nested form preserves the affine structure for tile-size calculations.

Affine indexing in couplings. A nested list inside a coupling entry denotes affine (summed) indexing. For example, [`Y1`, `R1`] in `int_in_coupling[0]` indicates activation indexing of the form $y_1 + r_1$, matching the sliding-window access pattern of a convolution. Optional stride dictionaries (`in_strides`, `w_strides`, etc.) allow each contributing

dimension to carry a multiplicative coefficient, generalising the model to strided convolutions (cf. Section 2.2.3).

Example: two-layer convolution. The coupling for a two-layer fusion ($F = 2$) provides a concrete illustration:

$$
\begin{aligned}
&\texttt{dims} && : [\,C_0,\, Y_0,\, X_0,\, R_0,\, S_0,\, Z_0,\, C_1,\, R_1,\, S_1,\, Z_1,\, P,\, Q\,] \\
&\texttt{in\_coupling} && : [\,C_0,\, [Y_0, R_0],\, [X_0, S_0]\,] \\
&\texttt{w\_coupling} && : \{0 : [\,Z_0, C_0, R_0, S_0\,],\;\; 1 : [\,Z_1, C_1, R_1, S_1\,]\} \\
&\texttt{int\_out\_coupling} && : \{0 : [\,Z_0,\, Y_0,\, X_0\,]\} \\
&\texttt{int\_in\_coupling} && : \{0 : [\,C_1,\, [P, R_1],\, [Q, S_1]\,]\} \\
&\texttt{out\_coupling} && : [\,Z_1,\, P,\, Q\,]
\end{aligned}
$$

Twelve dimensions, five operand categories, and the aliasing between `int_out`[0] and `int_in`[0] is immediately visible: $Z_0 \leftrightarrow C_1$ (channel aliasing), $(Y_0, X_0) \leftrightarrow ([P, R_1],\, [Q, S_1])$ (spatial aliasing).

The pattern generalizes mechanically: adding a layer appends six new dimensions (Z_i, C_i, R_i, S_i, Y_i, X_i for intermediate layers, or P, Q for the last), one new `w_coupling` entry, and one new pair of intermediate coupling entries.

5.2.2. Architecture Specification

An architecture in FactorFlow is an ordered list of *levels*, from outermost (e.g. DRAM) to innermost (Compute). Three level types are supported:

- **MemLevel**: a storage level characterized by capacity (bytes), per-access energy (picojoules per byte), separate read and write bandwidths (bytes per cycle), a bypass list, and optional factors and dataflow constraints.
- **FanoutLevel**: a spatial unrolling level with a fixed mesh size and the dimensions across which data is distributed.
- **ComputeLevel**: the innermost functional unit performing the MAC operation, with a fixed compute energy and latency (cycles per operation).

Five-category bypasses. The single-layer FactorFlow uses three bypass tags (`in`, `w`, `out`). The multi-layer extension adds `int_in` and `int_out`, enabling per-level control over which intermediate operands are stored. A level's bypass list specifies which operand categories are *not* stored at that level, equivalently, which operands "pass through" without incurring local access energy.

As an example, the DepFiN architecture specifies:

Level	Bypassed operands
DRAM	`int_in`, `int_out`
FeatureMemory	`w`
WeightMemory	`in`, `int_in`, `int_out`, `out`
AccReg	`in`, `w`, `int_in`

Critically, both `int_in` and `int_out` bypass DRAM: intermediate activations never leave the chip. They are stored exclusively in the FeatureMemory (FMEM), achieving the inter-layer data reuse that motivates fusion (Section 3.3.1).

Dataflow and factors constraints. Each `MemLevel` can specify:

- A `dataflow_constraints` list, fixing the loop ordering at that level.
- A `factors_constraints` dictionary, fixing the number of iterations of specified dimensions at that level.

The method `enforceFactorsConstraints` distributes the constraint factors automatically upon initialization, moving the required prime factors from the innermost level (where all factors start) to the constrained level.

5.2.3. Mapping Specification

A mapping in FactorFlow is defined by two pieces of information at every level of the architecture:

1. The **factors allocation**: the prime-factorized tiling factors assigned to each dimension at that level, determining how many iterations of each dimension execute there (cf. Section 2.4.1).
2. The **dataflow**: the ordering of dimensions at that level, determining the loop nesting and hence which operand enjoys innermost reuse (cf. Section 2.4.1).

The product of a dimension's factors across all levels must equal the dimension's full size (completeness), and the tile sizes at each level must respect the capacity and equality constraints attached to that level (feasibility).

For fused mappings, the factors allocation spans all $|\mathcal{D}|$ dimensions, both aliased and decoupled, and the producer–consumer pruning of Section 5.1.4 adds an additional feasibility filter on top of the standard capacity constraints.

Per-layer stationarity within a single loop nest. A key consequence of aliasing and decoupling (Sections 5.1.1–5.1.2) is that the unified dataflow implicitly encodes a *different mapping*, and hence a different *stationarity*, for each layer, all within a single loop nest.

Stationarity (Section 2.3.3 is determined by which dimensions iterate at a given level: an operand is stationary when the innermost loops at that level are orthogonal to it. Because decoupling makes the per-layer dimension sets disjoint ($\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$, Section 5.1.2), the dataflow ordering at any level is *filtered per layer*: at each level only the dimensions in $\mathcal{D}_i$ are visible to layer i. Within the unified ordering, each layer therefore "sees" its own sub-sequence of loops, and the innermost element of that sub-sequence determines that layer's stationarity at that level.

For instance, if we consider a two-layer fusion where the dataflow at a memory level orders the dimensions as $[\ldots, C_1, X_0, \ldots]$ (from outermost to innermost). Layer 0 sees X_0 as the innermost relevant loop: the weights $W_0[\, z_0, c_0, r_0, s_0 \,]$ do not depend on X_0, so W_0 is stationary. Layer 1 sees C_1 as its innermost relevant loop: the output $\mathrm{Out}[\, z_1, p, q \,]$ does not depend on C_1, so the output is stationary.

If we consider a different ordering, e.g. $[\ldots, Z_1, X_0, \ldots]$, would make W_1 vary on the innermost layer-1 loop (Z_1 indexes W_1), yielding an *input-stationary* dataflow for layer 1 while layer 0 remains output-stationary. The same unified nest thus accommodates layer 0 in output-stationary mode and layer 1 in input-stationary mode, no separate per-layer scheduling is needed.

This property follows directly from the modeling choices described earlier: aliasing gives each intermediate tensor its own set of concrete loop variables (Section 5.1.1), and decoupling guarantees that these per-layer variables are disjoint (Section 5.1.2). The unified dataflow is therefore strictly more expressive than F independent single-layer mappings laid out in sequence: it can reproduce any combination of per-layer stationarities while also capturing the inter-layer data dependencies through the producer–consumer pruning constraint.

Initialization sequence. Mapping construction follows a deterministic five-step sequence:

1. `initFactors`: all prime factors of every dimension are placed on the innermost level.

2. `enforceFactorsConstraints`: factors are moved outward to satisfy each level's equality ($=$) and lower-bound ($\geq$) factors constraints, using `moveFactor` (Section 5.3.2) with constraint checking temporarily disabled.

3. `setupBypasses`: for each bypassed operand, the last non-bypassing `MemLevel` receives a pointer to the chain of bypassed levels it must serve, enabling its `MOPs()` method to account for the extra traffic.
4. `setupSpatialLevelPointers`: each `MemLevel` is linked to the spatial levels (fanouts and compute) that follow it.
5. `checkFactorsConstraints`: a final global validation ensures the entire architecture satisfies all constraints.

After initialization, the resulting mapping, if and only if feasible, serves as the starting point for the map-space exploration engine.

5.3. Implementation

This section provides an overview of our codebase, focusing on the modules most affected by the multi-layer extension.

5.3.1. Code Structure

The implementation consists of approximately 5 500 lines of Python spread across ten principal modules:

Module	Lines	Responsibility
`factors.py`	742	`Coupling`, `Factors` (prime-factor dictionaries), `Shape` classes
`levels.py`	2192	`MemLevel` (incl. ∼700-line `MOPs()` method), `FanoutLevel`, `ComputeLevel`
`arch.py`	857	`Arch`: `moveFactor`, `validate_mapping_heuristic`, `enforceFactorsConstraints`
`cost_model.py`	707	`updateStats`, `Wart`, `EDP`, `Energy`, `Latency`
`engine.py`	115	Mapper entry point, multithreading orchestration
`settings.py`	187	Global configuration flags (`SEQUENTIAL_LAYER_EXECUTION`, `FULLY_CACHED`, etc.)
`pruning_check.py`	289	Standalone pruning validator for unit testing
`main_cli.py`	289	Command-line interface for specifying architecture and workload files
`prints.py`, `utils.py`	—	Output formatting and utility functions

The most substantial changes for multi-layer support are concentrated in `factors.py` (five-category `Coupling`), `levels.py` (per-layer MOPs decomposition), `arch.py` (feasibility pruning, per-layer constraint management), and `cost_model.py` (per-layer energy and latency).

Pre-defined coupling instances in `computations.py` allow workloads up to 10-layer fusions to be constructed by importing the corresponding coupling object and pairing it with a `Shape` specifying the dimension sizes. Larger fusions can be created programmatically by extending the same pattern.

5.3.2. The `moveFactor` Primitive

The map-space exploration in FactorFlow is built on a single primitive:

```
moveFactor(src, dst, dim, factor, amount)
```

This method transfers $\texttt{factor}^{\texttt{amount}}$ iterations of dimension `dim` from level `src` to level `dst`. The operation proceeds in six steps:

1. **Remove** the factor from the source level's prime-factor dictionary. If the source does not hold enough factors, the move fails immediately (returns `False`).

2. **Check source constraints.** If the removal leaves the source level in violation of a capacity or equality constraint, the factor is restored and the move fails.
3. **Add** the factor to the destination level.
4. **Update tile sizes.** For every level between source and destination, the tile size for `dim` is multiplied (or divided) by $\texttt{factor}^{\texttt{amount}}$, reflecting the changed iteration count.
5. **Check constraints on all affected levels.** If any level between source and destination (inclusive) violates its constraints after the tile-size update, the entire operation is rolled back: factor restored to source, removed from destination, tile sizes reverted.
6. **Feasibility pruning** Basing on the axis of the dimension that is changed, the corresponding full multi-layer producer–consumer pruning (Section 5.1.4) is evaluated on the new resulting mapping. If the axis producer–consumer pruning constraints fail for any layer, the move is fully rolled back.

The rollback guarantee ensures that every call to `moveFactor` is *atomic*: the architecture is either updated to a new valid state or left entirely unchanged. This property makes it safe to use `moveFactor` as the exploration step in hill-climbing, breadth-first, or any other local-search mapper without fear of accumulating partial state.

The mapper iterates through dimensions and levels, attempting `moveFactor` calls and evaluating the cost model after each successful move. Different mapper variants (*breadth-first_ local, linear, quadratic, exponential, hybrid*) differ in how they select which moves to try and how deeply they explore, but all rely on `moveFactor` as their sole mutation operator.

5.4. Cost-Model Extensions

The single-layer cost model (Section 2.6) computes three quantities: memory operations (MOPs), energy, and latency. Each must be generalized to handle $F > 1$ layers with five operand categories.

5.4.1. Per-Layer Memory Operations

The entry point `updateStats` traverses the architecture *top-to-bottom* (from DRAM toward Compute). At each `MemLevel`, the method `MOPs()` returns the *per-instance* read and write counts for every operand category:

Return value	Description
`in_reads`	Reads of the global input
`per_layer_w_reads`$[i]$	Reads of weight tensor W_i
`per_layer_int_in_reads`$[i]$	Reads of A_i as consumer input
`per_layer_int_out_reads`$[i]$	Partial-sum reads of A_i (accumulation)
`per_layer_int_out_writes`$[i]$	Partial-sum writes of A_i
`out_reads`	Partial-sum reads of the global output
`out_writes`	Writes of the global output

These per-instance counts are then **scaled** to total counts by multiplying each layer's contributions by the product of temporal and spatial iterations relevant to that layer:

$$\sigma_l^{(i)} \;=\; \tau_l^{(i)} \;\cdot\; s_l^{(i)}, \tag{5.9}$$

where $\tau_l^{(i)} = \prod_{k>l} \prod_{d \in \mathcal{D}_i} f_{k,d}$ is the product of temporal iterations of all dimensions relevant to layer i at levels above l, and $s_l^{(i)}$ is the analogous spatial iteration count accumulated from fanout levels. The set $\mathcal{D}_i$ contains only the dimensions coupled to layer i's operands, for example, layer 0 is influenced only by $\{C_0, Y_0, X_0, R_0, S_0, Z_0\}$.

Per-layer dimension filtering. At each level, a `dataflow_per_layer` dictionary is constructed, filtering the level's full dataflow to retain only the dimensions relevant to each layer. This filtering ensures that a dimension belonging solely to layer j does not inflate the iteration count attributed to layer i. The method `fullLayerProduct` computes the product of factor iterations for a specific layer at a specific level using exactly this filtered dimension set.

Bypass propagation. When an operand bypasses a level, its reads and writes at that level are zero. The level *above* the bypass chain is responsible for supplying data to the level *below*; FactorFlow models this by carrying "last reads" accumulators that propagate the most recent non-bypassed read count downward. For intermediate operands, this mechanism applies per-layer: each `per_layer_int_out_reads`$[i]$ and `per_layer_int_in_reads`$[i]$ is propagated independently through the bypass chain.

Drain handling. Writes from a lower level to a higher level (drains) are modeled symmetrically: when a level is not bypassed for an operand, the reads from the level above become writes into this level. The `FREE_DRAINS` setting controls whether drains contribute to energy and latency; when enabled, drain reads are assumed to be zero-cost.

5.4.2. Energy Model

The energy weighted MOPs (WMOPs) metric (cf. Equation 2.12) is extended to a per-layer decomposition:

$$\text{WMOPs}_i \;=\; \sum_{l \in \text{levels}} e_l \cdot \left(R_l^{(i)} + W_l^{(i)}\right), \tag{5.10}$$

where $R_l^{(i)}$ and $W_l^{(i)}$ are the total reads and writes at level l attributable to layer i, and e_l is the per-access energy of level l. For a boundary layer ($i = 0$ or $i = F-1$), $R_l^{(i)}$ includes the global input or output contributions; for intermediate layers, it includes only the corresponding intermediate and weight operands.

The total energy is:

$$\text{WMOPs} \;=\; \sum_{i=0}^{F-1} \text{WMOPs}_i \;+\; \text{WMOPs}_{\text{compute}}, \tag{5.11}$$

where $\text{WMOPs}_{\text{compute}}$ accounts for the MAC energy across all layers. The per-layer decomposition is valuable for identifying which layer dominates energy consumption and, in future work, for guiding per-layer mapping optimization.

5.4.3. Latency Model

Latency is computed *bottom-to-top*, starting from the Compute level and propagating upward through the memory hierarchy. At the Compute level, each tile executes in a fixed number of cycles (typically 1 for a single MAC). At each `MemLevel` above, the model determines whether the level is **compute-bound** or **memory-bound** on a per-layer basis.

Per-layer ideal bandwidth. For each layer i at level l, the ideal read bandwidth is:

$$b_{l,\text{rd}}^{(i)} \;=\; \frac{R_l^{(i)} + D_l^{(i)}}{\sigma_l^{(i)} \cdot T_l^{(i)}}, \tag{5.12}$$

where $D_l^{(i)}$ is the drain traffic, $\sigma_l^{(i)}$ is the scaling factor from Equation 5.9, and $T_l^{(i)}$ is the number of compute cycles per tile for layer i at level l (accounting for any stalls introduced by lower levels). The write bandwidth $b_{l,\text{wr}}^{(i)}$ is defined analogously using fills and updates.

Stall detection. If $b^{(i)}_{l,\mathrm{rd}} > B_{l,\mathrm{rd}}$ (the level's physical read bandwidth in bytes per cycle), the level is memory-bound for layer i's reads: the actual read-plus-drain latency becomes

$$\Lambda^{(i)}_{l,\mathrm{rd}} \;=\; \frac{R^{(i)}_l + D^{(i)}_l}{\sigma^{(i)}_l \cdot B_{l,\mathrm{rd}}}. \tag{5.13}$$

Otherwise the level is compute-bound and $\Lambda^{(i)}_{l,\mathrm{rd}} = T^{(i)}_l$. The same logic applies to the write side using fills and updates. The per-layer per-level latency is:

$$\Lambda^{(i)}_l \;=\; \max\!\big(\Lambda^{(i)}_{l,\mathrm{rd}},\ \Lambda^{(i)}_{l,\mathrm{wr}}\big), \tag{5.14}$$

and the stall cycles contributed by level l for layer i are $\Lambda^{(i)}_l - T^{(i)}_l$.

Sequential layer execution. Under the `SEQUENTIAL_LAYER_EXECUTION` model the per-level latency is the *sum* of per-layer latencies:

$$\Lambda_l \;=\; \sum_{i=0}^{F-1} \Lambda^{(i)}_l. \tag{5.15}$$

This models the fact that the PE array processes one layer at a time in a time-multiplexed fashion. The total latency is then the maximum across all levels:

$$\Lambda \;=\; \max_{l \in \mathrm{levels}} \Lambda_l. \tag{5.16}$$

Latency for single buffered memory levels. A level is single buffered when it cannot performs both reads and writes at the same time, in this situation, the latency of the level is not anymore the maximum of latency due to the reads and to the writes, but the sum. For instance, the FeatureMemory (FMEM) level in the DepFiN architecture is a single buffered memory level: because it is the on-chip buffer that stores all activations (input, intermediates, output), its read and write latencies are *summed* rather than taking the maximum:

$$\Lambda^{(i)}_{\mathrm{FMEM}} \;=\; \Lambda^{(i)}_{\mathrm{FMEM,rd}} + \Lambda^{(i)}_{\mathrm{FMEM,wr}}. \tag{5.17}$$

This reflects the observation that, for the access patterns of fused execution on DepFiN, the read and write traffic to FMEM cannot overlap fully. The resulting per-layer latency is then $\max(\Lambda^{(i)}_{\mathrm{FMEM}},\ T^{(i)}_l)$, preserving the guarantee that latency is never below the ideal compute time.

Compound metrics. The EDP (Energy-Delay Product) and Wart (weighted area-time) metrics defined in Section 2.6.4 apply unchanged; they simply use the extended WMOPs and latency values computed above.

5.4.4. Limitations

The current implementation has several limitations that constrain its applicability:

1. **No automated mapper for fused workloads.** The map-space exploration engine (`engine.py`) currently supports automated search only for single-layer mappings. For multi-layer fused workloads, is performed the evaluation of the single mapping described through the architecture definition file. Extending the mapper to automatically explore fused mappings, while respecting the pruning constraints, is the primary direction for future work.
2. **No inter-layer parallelism.** The `SEQUENTIAL_LAYER_EXECUTION` flag is always active: all layers share the same PE array and execute one at a time. Architectures that overlap computation of different layers (e.g. pipeline parallelism) are not tested.

6 | Validation

6.1. Introduction

The purpose of this chapter is to demonstrate that the analytical framework we developed accurately models the complex behavior of layer fusion and hardware execution on a real accelerator design. Establishing trust in the model is a prerequisite for the design-space exploration performed in the subsequent chapters: if the analytical predictions cannot be validated, any architectural insight drawn from them would be of limited value.

The validation is structured as follows. Section 6.2 summarizes the modeling methodology and the role of Accelergy as validated energy-estimation back-end. Section 6.3 describes the validation setup: the choice of hardware architecture, how its characteristics were extracted from the paper [10], how the mapping and dataflow were abstracted into the constraint model, and which workload was selected for comparison. Section 6.4 presents the validation results and discusses the agreement between the model and the reference design.

6.2. Methodology

As described in Chapter 5, the extension focus is to support multi-layer (fused) computation. The key extensions include:

1. Generalized loop-nest representation that unifies all fused layers into a single loop nest with per-layer dimensions (C_i, R_i, S_i, Z_i, Y_i, X_i) and inter-layer coupling constraints.

2. Memory-level bypass rules that distinguish between input, output, weight, intermediate input, and intermediate output operands, enabling precise modeling of on-chip data reuse across fused layers.

3. Cost model that computes per-level memory operations (MOPs), energy, latency (including bandwidth-limited stall cycles), and energy–delay product (EDP) for the

entire fused pipeline.

Not having access to DepFin fabricated silicon, energy per access at each level of the memory hierarchy is estimated using Accelergy [30], a plug-in-based framework validated to achieve roughly 95% accuracy for established spatial architectures.

6.3. Validation Setup

6.3.1. Motivation

The strongest form of validation for an analytical framework is comparison against a real hardware implementation, rather than against another high-level model. A hardware-level comparison tests not only the mathematical formulation of the cost model but also whether the architectural abstraction, memory sizes, bandwidth, bypass rules, spatial mapping, faithfully captures the behavior of the physical design.

For this reason, the validation target selected for this work is **DepFiN** [10], a depth-first CNN accelerator fabricated in 12 nm. DepFiN is a particularly suitable reference for several reasons.

First, DepFiN *natively supports layer fusion* as its primary execution mode.

The architecture was designed from the ground up for depth-first (fused) inference, with a split on-chip memory hierarchy (Feature Memory and Weight Memory) and a fixed dataflow that pipelines activations across up to ten consecutive convolutional layers without intermediate DRAM accesses. This makes it an ideal test case for the fusion extensions developed in this thesis.

Second, DepFiN has been adopted as an architectural benchmark by other analytical frameworks in the literature: DeFiNeS [20] and LoopTree [9] both model DepFiN as a representative example of a depth-first fused accelerator, which confirms its relevance and provides additional points of comparison.

Third, the DepFiN paper provides enough micro-architectural detail, PE array dimensions, memory sizes, wordline widths, spatial mapping strategy, and per-workload latency measurements as described in the following sections.

6.3.2. Architecture Extraction

Parameters taken from the Paper

The key architectural parameters extracted from the DepFiN paper [10] are summarized in Table 6.1.

Table 6.1: Architectural parameters extracted from the DepFiN paper [10].

Parameter	Value
Technology node	12 nm
Clock frequency	~930 MHz
PE array	$16 \times 128 = 2048$
Feature Memory (FMEM)	1056 KB (2 banks)
FMEM wordline	1056 bit
Weight Memory (WMEM)	524 KB (4 banks)
WMEM wordline	128 bit
DRAM type	LPDDR4
DRAM I/O bandwidth	96 bit = 12 B/cycle
Operand precision	8 bit
Accumulator precision	32 bit
Spatial mapping (cols)	Output width (Q / X_i)
Spatial mapping (rows)	Output channels (Z_i)

The 128 PE columns map onto the output spatial width dimension (Q for the final layer, X_i for intermediate layers), while the 16 PE rows map onto the output channel dimension (Z_i).

Assumptions and Limitations

Not all parameters required for a complete analytical model are available from the paper. The following assumptions were made:

1. **DRAM bandwidth.** The paper reports an aggregate external I/O interface of 96 bits (12 B/cycle) but does not provide a cycle-accurate DRAM bandwidth model; the reported latency values include unspecified stall effects from external memory.
2. **Feature Memory bandwidth.** We set the FMEM read bandwidth to 132 B/cycle and write bandwidth to 128 B/cycle, derived from the 1056-bit wordline and two-bank structure (1056 bit / 8 bit $\times$ 1 = 132 B per read access).
3. **Weight Memory bandwidth.** We set the WMEM bandwidth to 16 B/cycle for both reads and writes, derived from the 128-bit wordline (128 bit / 8 bit = 16 B per access).

4. **Energy values.** Per-access energy values are estimated by Accelergy at 12 nm rather than measured from the fabricated chip. The paper does not report per-component energy breakdowns, so direct comparison of energy is not possible.

Mapping Abstraction

DepFiN is not a programmable accelerator with a flexible mapping engine; it implements a *fixed* depth-first dataflow in hardware, where the data movement pattern is determined by the physical wiring between the Processing Element (PE) array, Feature Memory (FMEM), the specialized PE-FMEM interface and Weight Memory (WMEM).

In FactorFlow, this fixed dataflow is expressed through dataflow constraints and factor constraints at each memory level. The translation follows these principles:

1. **DRAM level.** All spatial dimensions (P, Q, and all Y_i, X_i) are tiled at DRAM, reflecting the fact that the accelerator processes one spatial tile at a time before moving to the next. The intermediate output width and the output width are divided into tiles of 128 pixels (matching the 128 PE columns), yielding a number of temporal iterations for Q equal to its total shape divided by 128. Meanwhile, all the intermediate output height and output height are divided into tile of 1 pixel height, all the rows iterated at this level. Intermediate operands(`int_in`, `int_out`) bypass DRAM entirely, enforcing the depth-first fusion invariant that no intermediate data touches external memory.

2. **Feature Memory (FMEM).** FMEM stores all non-weight operands: input activations, output activations, and intermediate activations between fused layers, while weights bypass this level entirely. Crucially, a *feature shifter* located between the PE array and the FMEM supports the reuse of neighboring features sent to the PEs. This mechanism allows incoming features to be temporally reused for a number of iterations equal to the kernel width S. Consequently, this interface reduces the required FMEM read bandwidth by a factor of S, which is essential to continuously feed the lower levels without introducing memory-bound stalls.

3. **Weight Memory (WMEM).** WMEM stores all weights for all 11 layers (10 fused + 1 output) simultaneously. All weight-related dimensions (C_i, R_i, S_i, Z_i) are constrained to their full values at this level, reflecting the fact that the complete filter set is pre-loaded before inference begins. All activation operands bypass WMEM.

4. **Spatial fanout levels.** The abstraction of having 16x128 PEs is performed thanks to spatial fanout levels that make possible the replication of all subsequent memory

levels.

In order to abstract the total availability of 2048 PEs for each layer during the respective layer computation, the abstraction performed in our model is that each of the 11 layers receives a dedicated pair of spatial fanout levels: 128 PEs of Spatial Columns distributes the output width, and 16 PE of Spatial Rows distributes the output channels, only one pair of spatial fanout is active at a given time instant, de facto are always used 16x128 PEs, not more. The factor constraints ensure that exactly 128 positions and 16 channels are processed in parallel per tile.

5. **Accumulation register.** A single register, replicated 2048 times, sits below all spatial levels and above the compute units. This register handles both output accumulation (across input channels) and intermediate-output storage (between fused layers). All spatial and output dimensions are constrained to factor 1 at this level, meaning it stores exactly one partial sum per PE at a time.

In DepFiN, the spatial tile (or patch) that must be computed during each DRAM iteration remains strictly constant across all fused layers. This uniformity is achieved through *buffered reuse* (Section 3.1), which allows the accelerator to compute only the newly required, non-overlapping portions of the intermediate feature maps. By caching the overlapping regions in the on-chip Feature Memory SRAM, this mechanism circumvents the cascading effect of inter-layer size dependencies formula (see Section 3.1.2) that would otherwise enlarge the intermediate area to compute. The trade-off is the requirement of an additional on-chip memory allocation (Equation 3.1) to store the buffered data necessary to prevent redundant recomputation.

These constraint-based encoding faithfully reproduces DepFiN's hardwired dataflow: there is no freedom to reorder the weight-related or spatial dimensions.

6.3.3. Workload Selection

The DepFiN paper [10] evaluates several workloads in its measurement results. For validation, we selected **Workload 2**, a 10-layer fusion pipeline operating on 720×1280 HD-resolution frames. Its layer structure is shown in Table 6.2.

This workload was chosen for its minimal external effects. Of the workloads evaluated in the DepFiN paper, Workload 2 has the fewest dependencies on implementation-specific optimizations that are not captured by the analytical model, such as power gating of unused PEs, dynamic clock scaling, or partial-array configurations described in the "Measurements and Metrics" section of [10]. Workloads that rely on these techniques would in-

Table 6.2: Workload 2 from the DepFiN paper: 10-layer fused pipeline at 720×1280 resolution. Layers 1–9 are identical 3×3 convolutions with 32 input and 32 output channels.

Layer	Resolution	Channels (in→out)	Kernel	MACs
L0	720×1280	3→32	3×3	796,262,400
L1	720×1280	32→32	3×3	8,493,465,600
L2	720×1280	32→32	3×3	8,493,465,600
L3	720×1280	32→32	3×3	8,493,465,600
L4	720×1280	32→32	3×3	8,493,465,600
L5	720×1280	32→32	3×3	8,493,465,600
L6	720×1280	32→32	3×3	8,493,465,600
L7	720×1280	32→32	3×3	8,493,465,600
L8	720×1280	32→32	3×3	8,493,465,600
L9	720×1280	32→32	3×3	8,493,465,600
L10	720×1280	32→16	1×1	471,859,200
			Total	**77,709,312,000**

troduce discrepancies unrelated to the modeling accuracy of the dataflow and cost model.

6.4. Validation Results

The validation run was executed as:

```
python3 main_cli.py architectures/depfin_arch.py
    architectures/10layer_computation.py
```

Table 6.3 reports the key results.

Table 6.3: Validation results: FactorFlow model vs. DepFiN ideal (compute-bound) reference for Workload 2.

Metric	DepFiN (ideal)	Our model
Total MAC operations	77,709,312,000	77,709,312,000
PE count	2,048 (16 × 128)	2,048 (16 × 128)
Spatial utilization	100 %	100 %
Ideal latency (cycles)	37,944,000	37,944,000
FMEM stall cycles (at regime)	0	0
WMEM stall cycles (at regime)	0	0
DRAM intermediate reads	0	0
DRAM intermediate writes	0	0

6.4.1. DRAM Bandwidth

As discussed in Section 6.3.2, the DepFiN paper does not provide enough information to construct a cycle-accurate model of the DRAM bandwidth. The reported aggregate I/O interface of 12 B/cycle (96 bits) does not account for LPDDR4 controller overhead, read/write turnaround penalties, or burst scheduling effects. For this reason, we do not attempt to validate the absolute total latency (which would include DRAM-induced stalls) and instead focus the comparison on the compute-bound latency, where our analytical predictions align perfectly with the reference, and the on-chip memory behavior, quantities that are fully determined by the architectural parameters known from the paper.

6.4.2. Data Residency and Intermediate Buffers

A central claim of the DepFiN design is that **intermediate activations between consecutive layers never touch DRAM**. The analytical model rigorously corroborates this property: as shown in Table 6.3, the total DRAM reads and writes for intermediate operands are both exactly zero. Crucially, the model demonstrates that the 1056 KB Feature Memory (FMEM) provides sufficient capacity to accommodate the entire buffered-reuse footprint (Section 3.3.2) for the 10-layer pipeline. Because this working set of intermediate activations fits strictly within the 1056 KB budget, no intermediate data is ever evicted, spilled to, or re-fetched from DRAM. This directly validates the architectural design point of DepFiN: the FMEM size is perfectly tuned for full depth-first fusion of the target workload, successfully preventing any intermediate recomputation.

Furthermore, the model successfully captures the critical temporal dynamics of the on-chip memory bandwidth. Due to the buffered-reuse caching policy, the accelerator avoids reading increasingly larger intermediate footprints (the cascading halo effect) from the FMEM, instead requiring a constant 1×128 spatial tile for each layer. However, the analytical engine reveals that even with this constant footprint, a standard FMEM interface with a read bandwidth of 132 would still inevitably trigger memory-bound stalls—for instance, computing a 10% latency overhead compared to the ideal compute-bound latency for Workload 2.

By explicitly modeling DepFiN's *feature shift register*, the framework demonstrates exactly how the required FMEM read bandwidth is decisively reduced. The model confirms that, with this intra-PE temporal reuse mechanism active, the Feature Memory introduces absolutely zero stalls during the regime execution. This precise alignment between the model's bandwidth predictions and the expected hardware behavior confirms that the proposed analytical framework is accurate, fully capable of evaluating both capacity

constraints and bandwidth bottlenecks in complex fused-layer dataflows.

7 | Experimental Results

This chapter, by exploring the complex design space of spatial accelerators, seeks to quantify the performance, energy, and memory trade-offs inherent in layer fusion. The investigation evaluate modified versions of the DepFiN and Eyeriss architectures across a diverse set of convolutional neural network workloads.

The chapter is organized into three main parts. First, the experimental setup is detailed, defining the selected hardware architectures, the targeted activation and weight-dominant workloads, and the comparative evaluation scenarios.

7.1. Experimental Setup

This section details the methodology and configurations used to evaluate the scheduling of multiple fused layers. Accordingly, Section 7.1.1 first defines the spatial architectures modeled in this study, specifically, modified variants of the DepFiN and Eyeriss v1 accelerators adapted for multi-layer execution. Next, Section 7.1.2 introduces the selected CNN workloads, covering both activation-dominant and weight-dominant networks to stress different aspects of the memory hierarchy. Finally, Section 7.1.3 outlines the evaluation scenarios, establishing a comparative framework between full fusion, partial fusion, and standard single-layer execution.

7.1.1. Architectures

The two spatial architectures selected for the experiments are DepFiN [10] and Eyeriss v1 [6]. Table 7.1 summarizes the key parameters of the *original* designs as published in the respective papers.

Both architectures were modeled in our model based on the parameters published in [6, 10]. However, because the spatial architectures must support a variety of fused CNN workloads, not only the single layer, several modifications were necessary. We therefore refer to the modified variants as **DepFiN-Like** and **Eyeriss-Like** throughout this chapter.

Table 7.1: Comparison of the original DepFiN and Eyeriss v1 architectures.

Parameter	DepFiN	Eyeriss v1
Technology node	12 nm	65 nm
Clock frequency	∼930 MHz	200 MHz
Dataflow	Depth-first (output-stationary)	Row-stationary
PE array	$16 \times 128 = 2048$	$12 \times 14 = 168$
On-chip memory	Split FMEM + WMEM	Unified GlobalBuffer
FMEM / GlobalBuffer size	1056 KB	128 KB
WMEM size	524 KB	—
PE registers	WeightReg (384), InputReg (24), AccumulationReg (32)	InputSpad (24), WeightSpad (384), PsumSpad (32)
Accumulator precision	32 bit	16 bit
DRAM type	LPDDR4	LPDDR4
DRAM bandwidth	12 B/cycle	8 B/cycle
FMEM read bandwidth	132 B/cycle	—
GB bandwidth	—	32 B/cycle
Spatial mapping (cols)	Output width (Q)	Output width (Q) + channels (M)
Spatial mapping (rows)	Output channels (M)	Filter (S) + channels (C, M)

DepFiN-Like

DepFiN is a depth-first CNN accelerator with a split on-chip memory hierarchy: a *Feature Memory* (FMEM) that stores input and output activations while bypassing weights, and a *Weight Memory* (WMEM) that stores weights while bypassing all activation operands. In fused (multi-layer) mode, intermediate activations between consecutive layers are kept entirely on-chip, DRAM bypasses the intermediate operands (`int_in`, `int_out`), so that inter-layer data never touches external memory (see Section 6.3.2 for further details). Each fused layer receives its own pair of spatial fanout levels (see Section 2.3.2): Systolic Array Columns (`SACols`) distributes the output spatial width (Q or X_i) across the PE columns, while Systolic Array Rows (`SARows`) distributes the output channels (M or Z_i) across the PE rows.

Compared with the original DepFiN design the following modifications were applied:

1. **Configurable tile size.** The original DepFiN fixes the output tile to a 1×128 pixel patch that fills the 128 PE columns exactly. Our version relaxes this constraint, allowing the output tile size to be set independently for each workload. This is essential for exploring the effect of tile size on energy, latency, and DRAM traffic across workloads with different spatial dimensions (see Section 7.2.1).

2. **Variable PE array dimensions.** Instead of the fixed 16×128 array, we explore

which results would give a different aspect ratios at a constant total PE count (Section 7.2.2).

3. **Mapping.** The depth-first dataflow is preserved: weights are loaded into WMEM for all fused layers, and activations stream through FMEM tile-by-tile. When the tile size or PE array dimensions change across case studies, the mapping adapts accordingly.

Eyeriss-Like

Eyeriss v1 is a row-stationary accelerator with a unified *GlobalBuffer* (128 KB) that stores input activations and output partial sums while bypassing weights (weights are delivered directly from DRAM to the per-PE weight registers). In the original design each PE contains three register files: an *InputSpad* for input activations, a *WeightSpad* for filter weights, and a *PsumSpad* for partial-sum accumulation. The row-stationary dataflow is expressed through the spatial fanout: Systolic Array Columns (`SACols`) distributes output spatial positions (Q) and output channels (M) across the 14 PE columns, while Systolic Array Rows (`SARows`) distributes filter rows (S), input channels (C), and output channels (M) across the 12 PE rows. Each PE therefore processes a 1-D convolution row: filter weights remain stationary in the weight register while input activations stream through, and partial sums accumulate across input channels along the columns.

Compared with the original Eyeriss, the following modifications were applied:

1. **Intermediate register for fusion.** The original Eyeriss has no register to store near the PE the intermediate activations. We add a fourth per-PE register file, the *IntermediateOutputRegister*, which stores intermediate activations produced by one layer and consumed by the next. This register bypasses all other operands (inputs, weights, final outputs) and, together with the DRAM bypass of `int_in`/`int_out` and to the dataflow, enables depth-first layer fusion on Eyeriss (see Section 7.1.3).

2. **Variable PE array dimensions.** With only 168 PEs (14×12), each PE must process a large share of the workload, primarily regarding the weights. The row stationary dataflow constraint the weight register to hold all the weights contemporaneously, for weight-dominant networks such as VGG16 and ResNet18, this forces the per-PE weight register to hold ten of thousands of bits, which is physically impractical. We therefore explore four total PE counts: 2 048, 4 096, 8 192, and 16 384. Going above 16 384 would exceed realistic budgets for low-power edge accelerators; going below 4 096 pushes the required weight-register size to several thousand bytes per PE, making the design infeasible. The sensitivity of register sizes and EDP to

the PE count and aspect ratio is analyzed in Sections 7.2.3–7.2.5.

3. **Register sizing.** Because the Eyeriss feasibility problem is multi-dimensional (InReg, WReg, IntermediateReg, and OutReg must all be satisfiable simultaneously for each workload and PE configuration), the per-PE register sizes are treated as experimental variables. Case studies CS1–CS3 sweep WReg, IntReg, and OutReg in turn while searching for the minimum feasible PE configuration at each size (Section 7.2.3, 7.2.4).

4. **Mapping.** The row-stationary dataflow is preserved, the mapping adapts when tile sizes or PE dimensions change across experiments.

Table 7.2: Summary of modifications applied to the original architectures.

Aspect	**DepFiN → DepFiN-Like**	**Eyeriss → Eyeriss-Like**
PE array	16 × 128 (fixed) → variable rows/cols	12 × 14 (fixed) → 2 048–16 384 total PEs
Tile size	Fixed 1 × 128 → configurable per workload	Determined by mapping → configurable per workload
Registers	Unchanged default sizes	Added IntermediateOutputReg (320 bytes); WReg/InReg/OutReg sizes swept as variables

Energy Model

All per-access energy values are derived from Accelergy using the CACTI plug-in for SRAMs, and the Neurosim plug-in for register files [30]. For the off-chip DRAM energy we adopt the value of 160 pJ / byte used by DepFiN [10], which is itself derived from [12]. Table 7.3 reports the energy per byte at each level of the memory hierarchy for a configuration used in case studies of each architecture.

Two observations follow from the table. First, DRAM access energy dominates the on-chip hierarchy. Because layer fusion aims to eliminate DRAM traffic for all intermediate activations (Section 7.1.3), the energy saving from fusion scales directly with this DRAM-to-on-chip ratio: the higher the assumed DRAM cost, the more fusion performs better.

Second, the Eyeriss register hierarchy is notably more expensive than DRAM-relative to the DepFiN case. In particular, the WeightRegister at 1 200 bytes costs 5.79 pJ / byte, which is already 0.036× the DRAM cost. Full fusion on Eyeriss requires very large per-PE registers to hold the state of all fused layers simultaneously, so each MAC operation

Table 7.3: Energy per byte of access at each memory-hierarchy level (pJ / byte), estimated by Accelergy at 12 nm. DepFiN config: FMEM = 568 KB, PE 16×128. Eyeriss config: GB = 128 KB, PE 512×32, WReg = 1200, InReg = 400, IntReg = 350, OutReg = 64.

DepFiN-Like		**Eyeriss-Like**	
Level	pJ / byte	Level	pJ / byte
DRAM	160.00	DRAM	160.00
FMEM	0.71	GlobalBuffer	0.24
AccReg	0.09	WeightReg	5.79
Compute (MAC)	0.10	InputReg	1.97
		IntermediateReg	1.73
		OutputReg	0.36
		Compute (MAC)	0.08

incurs a substantial register-energy overhead. This overhead can outweigh the DRAM savings, causing full fusion to consume *more* total energy and having a *higher* EDP than the unfused baseline despite eliminating nearly all DRAM traffic (Section 7.3).

7.1.2. Workloads

Two classes of CNN workloads were selected for the experiments: *activation-dominant* and *weight-dominant* (Table 7.4). The rationale for including both classes is that layer fusion primarily eliminates expensive DRAM reads and writes of intermediate activations, which would normally be transferred to and from external memory in single-layer or layer-by-layer processing (see Section 7.1.3). Activation-dominant workloads, characterized by large spatial dimensions and small channel counts, should therefore benefit the most from fusion, because intermediate activations constitute the dominant fraction of off-chip traffic. It is equally important, however, to evaluate fusion on weight-dominant workloads, where large channel counts and shrinking spatial dimensions cause weight-related DRAM traffic to outweigh activation traffic, making the gains from eliminating intermediate transfers less straightforward.

Two of the four workloads, VGG16 and ResNet18, contain stride-2 downsampling layers. When fusing across stride boundaries, the input tile size of a subsequent layer must account for the cumulative stride of all preceding layers (e.g. a 7×7 output tile at the end of ResNet18 requires a 112×112 input tile at the first layer), which increases the on-chip memory pressure for the fused workload.

Table 7.4: Summary of the four CNN workloads used in the experiments.

Workload	Type	Fused layers	Input size	Max channels	Stride
FSRCNN [7]	Activation-dom.	8	960×540	56	None
MC-CNN [17]	Activation-dom.	4	1242×376	32	None
VGG16 [4]	Weight-dom.	13	224×224	512	2 (4 layers)
ResNet18 [11]	Weight-dom.	17	224×224	512	2 (4 layers)

FSRCNN [7] is a super-resolution network composed of 8 convolutional layers with mixed filter sizes (5×5, 1×1, 3×3). The channel count remains small throughout (maximum 56), while the spatial resolution stays constant at 960×540 with no stride layers, making intermediate activation volumes very large relative to weight volumes.

MC-CNN [17] is a stereo-matching network with 4 homogeneous convolutional layers, all using 3×3 filters and a uniform channel count of 32. The input resolution of 1242×376 is the largest among the four workloads, making it strongly activation-dominant.

VGG16 [4] comprises 13 convolutional layers, all with 3×3 filters, organized in 5 blocks. Channels grow from 64 to 512 while the spatial resolution shrinks from 224×224 to 14×14 through four stride-2 transitions (modeling the max-pool between blocks), yielding a cumulative stride of 16.

ResNet18 [11] contains 17 main convolutional layers. The first layer uses a 7×7 filter with stride 2; three additional stride-2 layers occur at stages 3, 4, and 5, for a cumulative stride of 16. Channels increase from 64 to 512 while the spatial resolution decreases from 56×56 (after the first layer) to 7×7.

7.1.3. Partial and Full Fusion

As discussed in Section 7.1.2, full fusion of an entire workload can be very demanding even for a fusion-specialized accelerator, because the on-chip memory must simultaneously accommodate the weights, activations, and intermediate data of all fused layers. To capture fusion benefits without requiring an architecture that can host the full workload at once, we also evaluate *partial fusion*, in which only a small group of consecutive layers (typically two or three) is fused into a single segment. Each segment is executed independently, and the total energy and latency are obtained by summing over all segments:

$$E_{\text{total}} = \sum_i E_i, \qquad L_{\text{total}} = \sum_i L_i, \qquad \text{EDP} = E_{\text{total}} \times L_{\text{total}}. \tag{7.1}$$

Example: FSRCNN. FSRCNN has 8 convolutional layers. Under 2-layer partial fusion the workload is split into four segments, each fusing two consecutive layers: L0 + L1, L2 + L3, L4 + L5, and L6 + L7. Each segment is mapped and executed independently on the same architecture, and the results are aggregated with Equation (7.1). In contrast, under full fusion all 8 layers are fused into a single execution.

Evaluation Scenarios

Because the architecture must be sized differently depending on the fusion level it supports, we define two evaluation scenarios:

Scenario A (full-sized architecture). The on-chip memories and registers are dimensioned to support *full fusion* of the entire workload, i.e. they are large enough to hold the weights and intermediate data of all fused layers simultaneously. Three configurations are compared on this same full-sized architecture: (i) *full fusion*, where all layers are fused into one execution; (ii)Sum of (Σ)*partial fusion*, where partial-fusion segments are executed independently and their results summed; and (iii)Sum of (Σ)*singles*, where every layer is executed individually. This provides a fair comparison, as all three strategies use identical hardware.

Scenario B (partial-sized architecture). The on-chip memories and registers are dimensioned only for the largest partial-fusion segment, resulting in a *smaller and less energy-intensive* architecture than Scenario A. Full fusion is not feasible on this reduced hardware, so only sum of partials fusion and sum of singles are compared. This scenario answers a practical question: how much the partial fusion will cost and, if full fusion was beneficial, is partial fusion still beneficial compared with single-layer execution on the same smaller chip?

Table 7.5 illustrates the memory sizing difference between the two scenarios for selected configurations.

Table 7.5: Example architecture sizing for Scenario A and Scenario B.

Configuration	Scenario A	Scenario B	Difference
Eyeriss, ResNet18 (512×32)	WReg = 902	WReg = 384	2.3× smaller
Eyeriss, VGG16 (512×32)	WReg = 1 200	WReg = 576	2.1× smaller
DepFiN, FSRCNN	FMEM = 576 KB	FMEM = 248 KB	2.3× smaller
DepFiN, ResNet18	WMEM = 10 738 KB	WMEM = 4 608 KB	2.3× smaller

7.2. Single-Architecture Sensitivity Analysis

This section presents sensitivity sweeps performed on each architecture in isolation, varying one design parameter at a time while keeping all others fixed. The goal is to understand how tile size, PE array dimensions, and register sizing affect energy, latency, and EDP for each workload before moving to the fusion comparison in Section 7.3.

7.2.1. DepFiN: Tile Size Sensitivity (CS1)

Motivation

In the DepFiN-Like architecture the output tile size determines how many output columns are computed in parallel across the PE array columns. An output tile of width t occupies t of the 128 available columns, so the number of temporal passes over the output spatial dimension Q at the DRAM level is $\lceil Q/t \rceil$. Reducing the tile size therefore multiplies the number of DRAM-level iterations, which has two consequences:

1. **Latency increases** almost linearly, because each additional pass requires a full traversal of all fused layers for that tile slice.
2. **Weight re-reads increase**: at every new tile pass the entire weight set must be re-fetched from the Weight Memory (WMEM) into the per-PE weight registers, since the weights must be available for each pass.

The Feature Memory (FMEM) read bandwidth is scaled proportionally to the tile size ($\mathrm{BW}_{\mathrm{scaled}} = \mathrm{BW}_{\mathrm{base}} \times t/128$) to reflect the reduced wordline utilization at smaller tiles, without this change, we would simply see an under utilization of the bandwidth. All other parameters are held constant: 16×128 PEs (2 048 total), FMEM = 1 056 KB, WMEM = 524 KB.

Remark on tile dimensionality. The sweep varies the tile *width* (output dimension Q), which is the dimension spatially mapped to PE columns. The choice of sweeping width rather than height is not a fundamental limitation: in both the DepFiN-Like and the Eyeriss-Like architectures the height dimension P is handled identically, as a purely temporal iteration at the DRAM level. A sweep over tile height, or over both dimensions jointly, would therefore require no modification to our model, but the results would not be efficient in both the architecture due to the native parallelization of dimensions.

Results

Figure 7.1 shows the EDP (log scale) as a function of tile size for each workload. The best output tile size width, marked with a gold star, is always the *largest* feasible tile, confirming that maximizing spatial parallelism in the column dimension is uniformly beneficial.

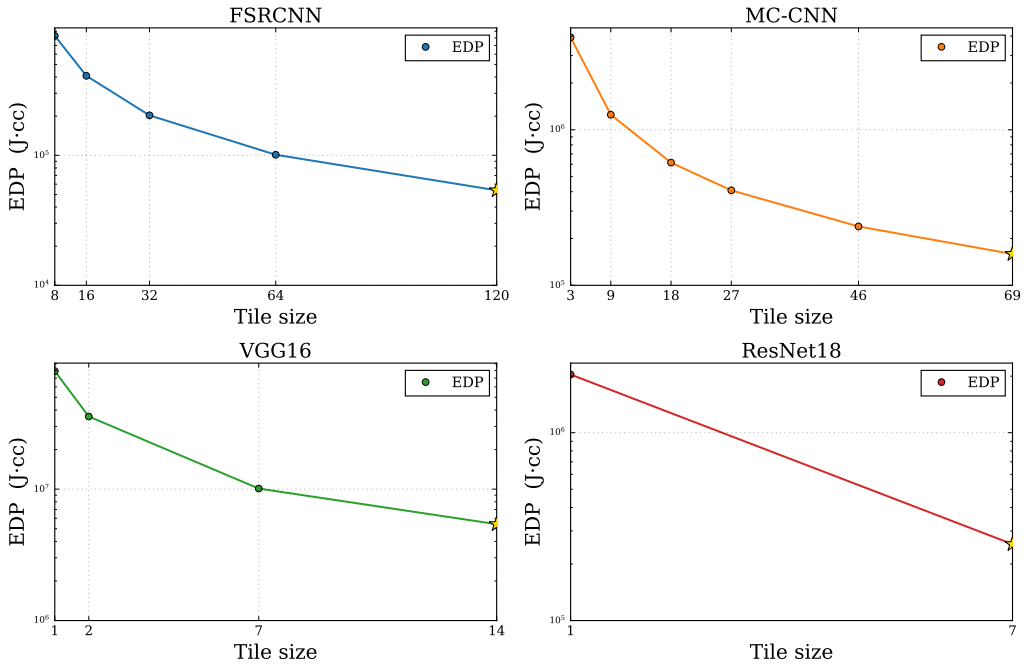

Figure 7.1: DepFiN CS1: EDP vs tile size for each workload The gold star marks the best (minimum-EDP) tile.

FSRCNN (activation-dominant, no stride). As a representative example, FSRCNN is swept from an output tile width of 120 (93.8 % PE utilization) down to tile width of 8 (6.2 %). The EDP degrades by $\approx 15\times$ driven almost entirely by latency, which increases by 15×. Energy, in contrast, rises by only +4 %. The mild energy increase is explained by the weight re-read mechanism: with an output tile width of 120 the Weight Memory is read 8.1×10^7 times, whereas at output tile width of 8 the same weights are re-fetched 1.2×10^9 times (15× more), because each of the additional DRAM passes triggers a full reload of the weight set from Weight Memory. DRAM reads and writes remain *identical* across all tile sizes, confirming that the degradation is internal to the on-chip hierarchy.

Weight-dominant workloads. VGG16 and ResNet18 exhibit a qualitatively similar trend, but with a larger energy increase (+25 % and +26 % respectively from best to worst

tile), due to larger weight volumes to be reloaded. The EDP degradation ranges from 8× (ResNet18, tiles $7 \rightarrow 1$) to 15× (VGG16, tiles $14 \rightarrow 1$).

Considerations. For all four workloads, the largest feasible tile size minimizes EDP in DepFiN. The dominant effect is latency: smaller tiles force more temporal iterations at the DRAM level. The effect of Energy is secondary and grows mainly through increased weight re-reads in the Weight Memory, but this effect is more pronounced for weight-dominant workloads.

7.2.2. DepFiN: PE Aspect Ratio (CS2)

Motivation

CS2 address a complementary question with respect to the precedent case studies: *given a fixed total PE budget, how should the array be shaped, i.e. how many rows versus columns, to minimize EDP?* CS2 fix the total number of PEs at 2 048 and varying only the aspect ratio, from the extremely column-heavy configuration $2 \times 1\,024$ to the extremely row-heavy configuration $1\,024 \times 2$. Memory sizes and register capacities are held constant at the per-workload values used throughout the DepFiN experiments.

Because rows map to output channels Z and columns map to tile width Q, shifting PEs from columns to rows simultaneously reduces the highest tile size available and increases the Z-parallelism. The tile size, in turn, is chosen for the experiment as the largest divisor of Q that fits within the available columns. A strongly column-heavy shape therefore wastes rows (few Z channels processed per pass, many re-reads of input activations from the Feature Memory), while a strongly row-heavy shape wastes columns (tiny tile, many DRAM-level passes, many weight re-reads from the Weight Memory).

The EDP curve is thus expected to be U-shaped, with a minimum at the aspect ratio that best balances the two effects.

Results

Figure 7.2 shows EDP (log scale, primary axis) and energy (μJ, linear scale, secondary axis) as a function of aspect ratio for each workload. The gold star marks the EDP-optimal configuration.

FSRCNN (activation-dominant). The EDP curve exhibits a broad minimum centered on 16×128, with the neighboring configurations 4×512 and 8×256 within 5 % and 3 % respectively. Moving toward the row-heavy end the EDP rises steeply: at 512×4,

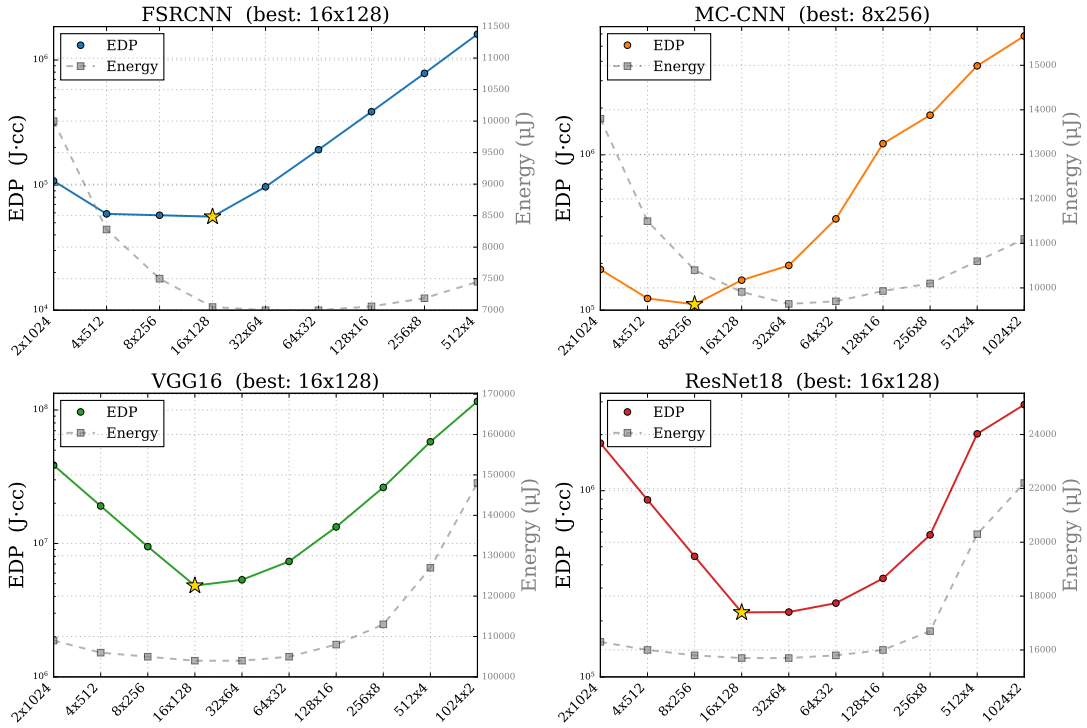

Figure 7.2: DepFiN CS2: EDP and energy vs. PE aspect ratio (total PEs = 2 048). The gold star marks the minimum-EDP configuration.

the increase is driven entirely by latency, which grows by 28× from 16 × 128 configuration to 512 × 4, as the tile shrinks and each additional DRAM pass reloads the full weight set. Energy, in contrast, varies by only ±20 % across the entire sweep, confirming that for activation-dominant workloads the dominant energy component, register-level operations, is largely insensitive to the array shape.

MC-CNN (activation-dominant). MC-CNN is the only workload tried whose optimum shifts away from 16 × 128 PE configuration: the best configuration is 8 × 256. The shift occurs because MC-CNN's output spatial dimension ($Q = 1\,242$) benefit from the wider tile that 256 columns provide, reducing DRAM-level passes beyond what 128 columns can achieve. At the column-heavy extreme (2 × 1 024) the EDP is 1.7× worse, while at the row-heavy extreme (1 024 × 2) it is 53× worse, making the U-shape strongly asymmetric: the penalty for too many rows far exceeds the penalty for too many columns.

Weight-dominant workloads (VGG16, ResNet18). Both workloads are optimized at 16×128. VGG16 EDP drops sharply from 2×1 024 to 16×128 with an 8× improvement, then rises again at 1 024 × 2 configuration, 24× worse than the optimum.

The left-hand descent is driven by latency: at 2 × 1 024 only two output channels are

mapped per pass, requiring many temporal iterations over Z, while at 16×128 the bottleneck layer's channel count (64) is covered in four passes. The right-hand ascent is driven by both latency and energy: as columns shrink below 128 the tile fragments, weight memory re-reads multiply, and energy increases by +42 %. ResNet18 mirrors this pattern, with the EDP at the row-heavy extreme 13× above the optimum.

Considerations. All four workloads produce a U-shaped EDP curve, confirming that both extremes of the aspect ratio are not suitable for these workloads. The optimum lies near 16×128 for three of the four workloads and at 8×256 for MC-CNN, whose large output spatial dimension rewards a wider tile. The curve is consistently asymmetric: but the rapidity of growing of EDP is determined by the activation or weight dominance of the workload.

7.2.3. Eyeriss: Weight Register Sizing (CS1)

Motivation

In the Eyeriss-Like architecture every PE stores its assigned weight filters in a local Weight Register (WReg). When multiple layers are fused, a single PE must hold the weight residuals for *all* fused layers simultaneously, that is, the weight iterations that are not spatially absorbed by the Systolic Array Rows (SARows) mapping. The WReg is therefore the largest per-PE register in case of Weight-Dominant workloads, and it acts as the binding constraint on feasibility: if the required weight footprint exceeds the WReg capacity, no valid mapping exists for that PE configuration.

The trade-off between WReg size and the minimum number of PEs can be expressed as:

- A higher total PE count means that more weight factors are spatially absorbed, enabling a *smaller* WReg.
- A smaller total PE count means that each PE must store more residual in its WReg, so it must to be *larger*, at the cost of having higher energy access cost.

This case study answers to the question: *what is the minimum weight register size that makes the fused mapping feasible at this given PE count?*

CS1 answers this question by sweeping the WReg size at a given PE configuration (rows × cols, up to 16 384 total PEs), in order to find the minimum WReg size feasible and the resulting energy and EDP.

The GlobalBuffer (128 KB), input register, intermediate register, and output register are

held constant throughout.

Results

Weight-dominant workloads (VGG16, ResNet18). Table 7.6 reports the minimum WReg size required at three PE budgets, together with the configuration chosen by the mapper and the resulting energy, latency, and EDP.

Table 7.6: Eyeriss CS1: Minimum WReg size for a given PE budget (VGG16 and ResNet18, full fusion). For each row the PE budget is fixed and the smallest feasible WReg is reported.

Workload	PE budget	Min WReg	Config	Energy (μJ)	Latency (cc)	EDP (J·cc)
VGG16	16 384	1 200	256×64	1.62×10^5	5.46×10^6	9.04×10^5
VGG16	8 192	2 400	128×64	2.47×10^5	5.68×10^6	1.42×10^6
VGG16	4 096	4 800	128×32	4.18×10^5	6.72×10^6	2.82×10^6
ResNet18	16 384	902	256×64	2.35×10^4	3.04×10^6	7.36×10^4
ResNet18	8 192	1 800	128×64	3.28×10^4	3.04×10^6	1.02×10^5
ResNet18	4 096	3 600	128×32	5.18×10^4	3.06×10^6	1.60×10^5

The table reveals a regular scaling law. For ResNet18 a designer with a 16 384-PE budget needs at least WReg = 902 bytes, and only two PE configurations out of the entire grid are feasible at that size. Halving the PE budget to 8 192 doubles the required register to WReg = 1 800, and halving again to 4 096 doubles it once more to WReg = 3 600.

As stated, fewer spatial rows leave more weight factors unabsorbed, and those residuals must be stored temporally in the register.

VGG16 obeys the same doubling rule ($1\,200 \rightarrow 2\,400 \rightarrow 4\,800$) at proportionally larger WReg values, reflecting its wider channels.

The energy consequence is substantial. For ResNet18, moving from the 16 384-PE point to the 4 096-PE point increases energy by 2.2×, because each weight access now touches a register that is 4× larger. Latency, however, stays nearly constant (less than 1% of difference), which indicates that the computation is not memory-bound (5.4.3) once a feasible mapping exists. The EDP therefore tracks energy closely, growing by 2.2× over the same range.

VGG16 shows both a steeper energy increase (2.6×) and a moderate latency penalty ($\approx 23\,\%$), compounding into a 3.1× EDP rise across the PE budget sweep. The latency degradation is driven by the middle layers (L2–L6, Z=128–256, 56×56–112×112), where

the constrained PE count increases weight traffic beyond the DRAM bandwidth capacity, causing memory stalls.

Figure 7.3 The horizontal axis is the PE budget and the vertical axis is the minimum WReg required. Reading the curve from left to right, one can directly read off the register size that a given area constraint demands.

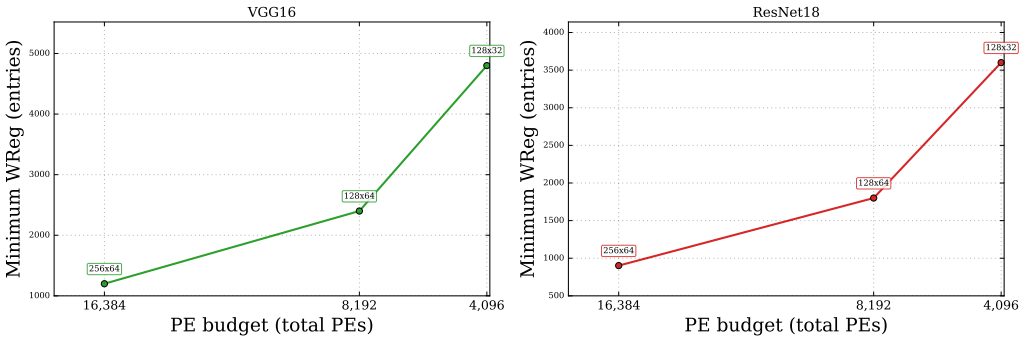

Figure 7.3: Eyeriss CS1: Minimum WReg required as a function of PE budget (VGG16 and ResNet18).

Activation-dominant workloads (FSRCNN, MC-CNN). For these workloads the weight register is not the binding constraint. FSRCNN and MC-CNN reaches feasibility for 1024 PEs at values of WReg lower than the value of default single layer Eyeriss (384 bits).

Energy still scales linearly with register size, but the effect is modest because the register is small in absolute terms.

Considerations. The weight register is a critical sizing knob for the Eyeriss-Like architecture under full fusion. For weight-dominant workloads, halving the PE budget roughly doubles the required WReg, and the larger register inflates energy in proportion, because every weight read access a physically larger SRAM structure. At the number of PEs tried, latency is less affected, so EDP tracks energy mainly. For activation-dominant workloads the register is non-binding and can be kept small without limiting feasibility.

7.2.4. Eyeriss: Activation Register Sizing (CS2, CS3)

Motivation

Besides the weight register examined in CS1, every PE in the Eyeriss-Like architecture contains three local storage structures: an *input register* (InReg), which buffers the first

layer's input activations; an *intermediate register* (IntReg), which buffers the activation values exchanged between consecutive fused layers; and an *output register* (OutReg), which holds the final-layer output before it is written back to the GlobalBuffer.

Of these three, only IntReg (CS2) is genuinely affected by fusion. The InReg sees exclusively the network's input, regardless of how many layers are fused: intermediate activations produced inside the fusion chain never pass through InReg, they are routed to IntReg instead. Similarly, the OutReg stores only the last fused layer's output channel residual, so its footprint does not grow with the number of fused layers.

We will analyze InReg and OutReg in a unique case study CS3.

Under full fusion the IntReg must accommodate the residual channel factors of *all* intermediate activations, i.e. the Z_i values that the systolic array columns (SACols) cannot absorb. The footprint therefore scales with the number of fused layers and with the width of their channel dimensions.

CS2 per Weight-dominant workloads (ResNet18, VGG16) :

The table reveals the same doubling rule observed for WReg in CS1.

Figure 7.4 the horizontal axis is the PE budget and the vertical axis is the minimum IntReg that makes the fused mapping feasible. Reading the curve from left to right, one can directly read off the register size that a given area constraint demands. The slope is consistent with the doubling rule: every 2× reduction in PE budget roughly doubles the minimum IntReg.

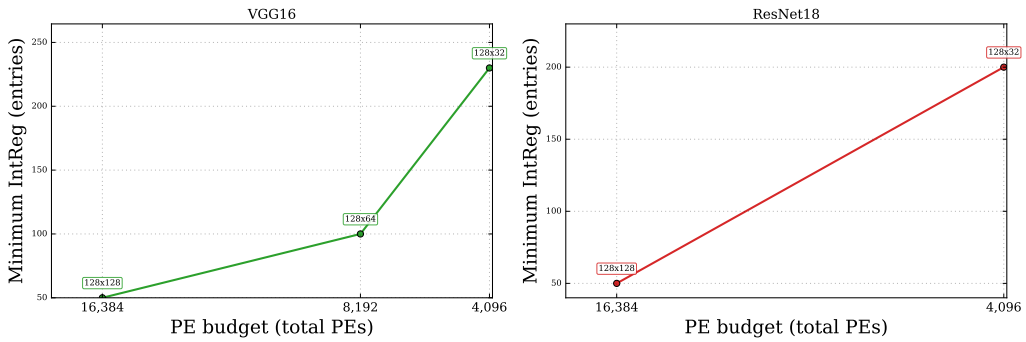

Figure 7.4: Eyeriss CS2: Minimum IntReg required as a function of PE budget (VGG16 and ResNet18, full fusion).

Figure 7.4 (top row) reports the energy consumption at the minimum-PE configuration for each IntReg size. Energy grows monotonically with IntReg capacity, as a larger register

incurs a higher per-read access cost. For ResNet18, the energy increases by 1.4× when scaling from IntReg = 50 to IntReg = 1 000. This is a relatively modest penalty compared to the 2.2× energy span caused by the WReg expansion in CS1, a difference that directly reflects the weight-dominant nature of the workload.

CS2 per Activation-dominant workloads (FSRCNN, MC-CNN). For these workloads the intermediate register is not a binding constraint. In both cases, even with a budget of PEs of 1024, the intermediate footprint is small (16) because these networks have few fused layers and narrow channels.

CS3 The output register stores only the last fused layer's channel residual, so its footprint is inherently smaller than that of IntReg or WReg. The original single-layer Eyeriss design uses OutReg with a capacity of 16 bytes, and this value turns out to be sufficient for full fusion as well at the PE budgets that WReg and IntReg already demand.

The curves lie well below the corresponding CS1 and CS2 curves at every PE budget, confirming that OutReg is the least binding of the three register types. For activation-dominant workloads (FSRCNN, MC-CNN) the output register has no effect on the PE count whatsoever: all tested PEs budget have could support the OutReg size of the Eyeriss single layer configuration.

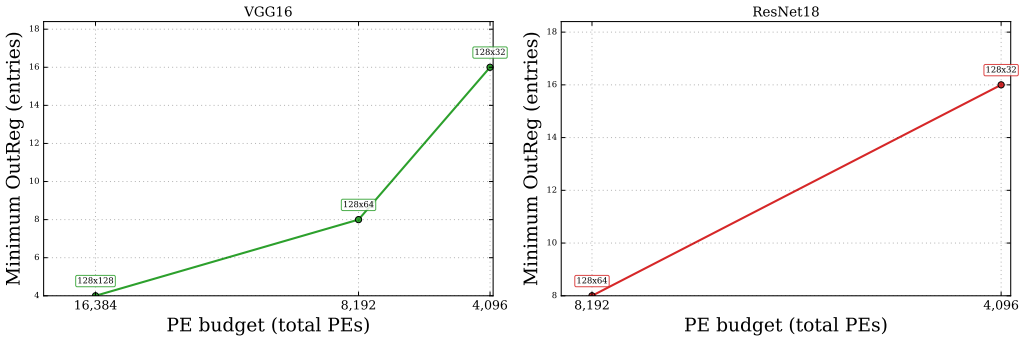

Figure 7.5: Eyeriss CS3: Minimum OutReg required as a function of PE budget (VGG16 and ResNet18, full fusion).

Analogous reasoning can be applied for Input register sizing.

Considerations The intermediate register follows the same PE-doubling-per-halving rule as the weight register, but its absolute footprint is roughly one order of magnitude smaller, so it becomes binding only at PE counts well below the WReg minimum. The output register is even less constraining: with OutReg with a capacity 16 bytes (the value used in the original single-layer Eyeriss), the fused mapping is already feasible at the PE

budgets imposed by WReg. The input register is unaffected by fusion too, because it handles only the first layer's input and its footprint remains the same as in single-layer execution.

7.2.5. Eyeriss: PE Array Aspect Ratio (CS5)

Motivation

Optimizing the PE array aspect ratio for EDP (see Section 7.2.2) is also essential in Eyeriss, because its spatial mapping distributes different loop dimensions across the two axes. Rows absorb weight-related factors (C, S, Z), whereas columns absorb output channel factors.

Therefore, a row-heavy array absorbs more weight factors spatially but forces larger channel residuals in the column direction, while a column-heavy configuration exhibits the inverse behavior. Ultimately, the optimal shape is governed by the relative size of the weight and activation dimensions in the fused workload.

To evaluate this trade-off, CS5 sweeps all power-of-two factorizations of a fixed PE budget (2,048 PEs for FSRCNN and MC-CNN; 16,384 PEs for VGG16 and ResNet18), tracking the resulting energy, latency, and EDP for every feasible mapping.

Results

ResNet18 (16 384 PEs). Table 7.7 lists the feasible configurations for ResNet18, with a weight register size fixed. Seven aspect ratios are feasible, from 256×64 to the extreme row-heavy $16{,}384 \times 1$. Latency is constant across all configurations, therefore the EDP ranking is determined entirely by energy. The best configuration is 256×64, and energy increases gently as the array becomes more row-heavy.

The total variation is only 2.6%, indicating that for ResNet18 the aspect ratio has a negligible impact once feasibility is achieved.

The underlying reason is that ResNet18 features deep channel dimensions (Z up to 512) that easily saturate a moderate column count. At 64 columns, the spatial mapping mechanism (`SACols`) effectively absorbs the most Z factors. Conversely, if the array is constrained to fewer than 64 columns, the mapper is forced to handle the excess Z dimensions temporally. This temporal folding increases the required capacity of the weight register to accommodate the larger residuals, which marginally raises the energy cost per access.

Table 7.7: Eyeriss CS5: PE aspect ratio sweep for ResNet18 (16 384 total PEs, full fusion). Best EDP marked with ⋆.

Config	Rows	Cols	Energy (μJ)	EDP (J·cc)
$256 \times 64 \star$	256	64	2.35×10^4	7.36×10^4
512×32	512	32	2.37×10^4	7.37×10^4
1024×16	1024	16	2.40×10^4	7.46×10^4
2048×8	2048	8	2.41×10^4	7.50×10^4
4096×4	4096	4	2.41×10^4	7.50×10^4
8192×2	8192	2	2.41×10^4	7.50×10^4
16384×1	16384	1	2.41×10^4	7.50×10^4

VGG16 (16 384 PEs). Seven power-of-two factorizations from 256×64 to 16,384×1 are feasible, all sharing the same latency. The best EDP is 512×32, tied with 256×64. Energy rises by 4.1% at 2,048 × 8 and plateaus thereafter, mirroring the ResNet18 behavior.

FSRCNN (2 048 PEs). FSRCNN exhibits a broader feasibility range, with nine power-of-two factorizations from 8 × 256 to 2,048 × 1 being feasible. The best EDP is the most column-heavy configuration, 8 × 256. Energy is 5.7% lower at 8 × 256 than at 64 × 32 and beyond, where it plateaus.

MC-CNN (2 048 PEs). Nine configurations are feasible from 8×256 to 2,048×1. The best EDP is 128 × 16, tying with 256 × 8 through 2,048 × 1, while 16 × 128 is within 0.2% and offers the lowest latency among competitive configurations. The most column-heavy configuration (8 × 256) has higher latency because only 8 rows cannot absorb enough weight factors spatially, forcing more temporal iterations. MC-CNN therefore favors a moderate aspect ratio that balances row absorption of weights with column distribution of channels.

Cross-workload comparison. Figure 7.6 presents the energy (top row) and EDP (bottom row) for all four workloads. Two patterns emerge. First, the weight-dominant workloads (VGG16, ResNet18) show nearly flat energy and EDP curves across all seven feasible aspect ratios, with total variation below 4.1% for VGG16 and 2.6% for ResNet18. Second, the activation-dominant workloads (FSRCNN, MC-CNN) display a mild preference for column-heavy configurations, where the Systolic Array Columns (SACols) mechanism can distribute channel factors more effectively. In all cases the EDP variation across feasible aspect ratios is modest ($< 6\%$ for FSRCNN, $< 4.1\%$ for VGG16).

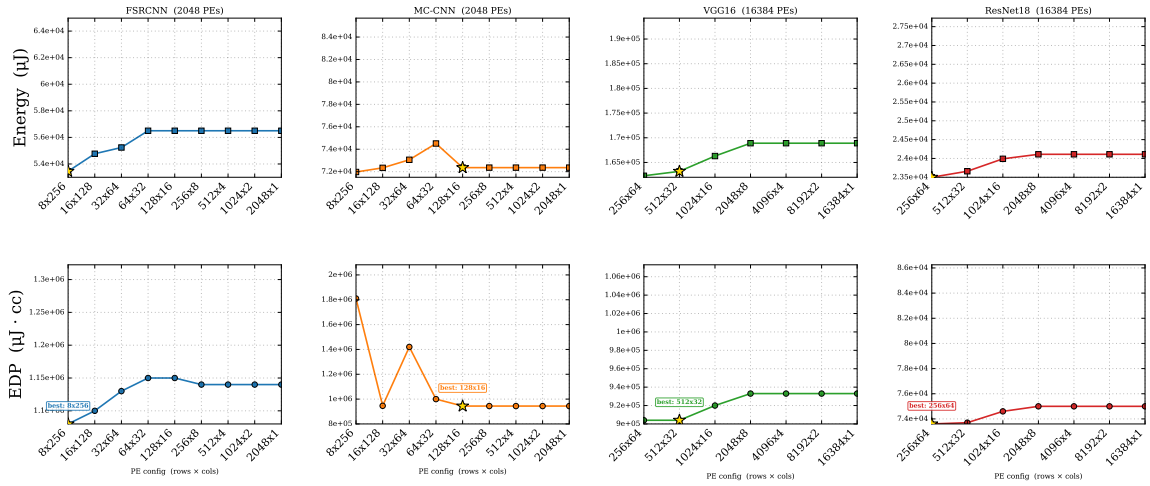

Figure 7.6: Eyeriss CS5: Energy (top) and EDP (bottom) vs. PE aspect ratio at fixed total PEs. Stars mark the best-EDP configuration for each workload.

Considerations Once the register sizes are set to satisfy the fused mapping constraints, the PE aspect ratio has limited influence on energy and EDP. Column-heavy configurations offer a small advantage for workloads with narrow channels, while weight-dominant workloads show nearly flat curves regardless of aspect ratio once registers are adequately sized.

7.3. Layer Fusion Analysis

This section evaluates the benefit of layer fusion by comparing fused, partial fused and non-fused execution on the same hardware architecture. Two scenarios are considered: Scenario A sizes the architecture for full fusion and runs all three modes (full, partial, single) on it; Scenario B sizes the architecture for partial fusion only, removing the full-fusion configuration to test whether a smaller design can still capture most of the fusion benefit.

7.3.1. Full-Sized Architecture: Fusion Mode Comparison (Scenario A)

Motivation

The case studies presented in Sections 7.2.3 through 7.2.5 focused on how individual architectural parameters, such as register sizes and PE aspect ratios, interact with the fusion mapping. This section shifts perspective and answers the question: *does layer fusion actually improve the end-to-end efficiency of a DNN accelerator on the same architecture instance, and under what conditions?*

To answer this question, we define three execution modes on the **same architecture instance**, which is sized to support full fusion of the entire workload (Scenario A):

1. **Full fusion.** All convolutional layers are fused into a single mapping. Intermediate activations are kept entirely on chip, reducing DRAM traffic to the network's input and final output only.
2. **Partial fusion (sum).** Consecutive layer groups are fused independently. The total cost is the sum of all fused evaluations. DRAM traffic is reduced for each group, but intermediate activations between non-fused layers still traverse DRAM.
3. **Single-layer (sum).** Each layer is mapped in isolation, and the total cost is the sum of all individual layer evaluations. Every intermediate activation is written to and read back from DRAM.

The architecture used for each workload is the **same** full-fusion configuration determined by the PE and memory sizing studies done (Table 7.8). Running partial and single-layer modes on a full-fusion-sized architecture provides a fair comparison because all three modes share identical hardware; only the inter-layer mapping strategy differs.

The following analysis is organized around three metrics, each normalized to the full-

Table 7.8: Architecture configurations used for Scenario A (full-fusion-sized). Memory sizes refer to the DepFiN feature-memory (fmem) and weight-memory (wmem), or the Eyeriss GlobalBuffer (GB) and per-PE weight register (WReg).

		DepFiN		Eyeriss	
Workload	PEs	fmem	wmem	GB	WReg
FSRCNN	16×128 / 128×16	576 KB	19 KB	128 KB	384
MC-CNN	8×256 / 256×8	522 KB	32 KB	128 KB	384
VGG16	16×128 / 512×32	568 KB	14 366 KB	128 KB	1 200
ResNet18	16×128 / 512×32	266 KB	10 738 KB	128 KB	902

fusion value, and concludes with the DRAM traffic reduction and a summary of the key findings.

Results

Energy (Figure 7.7). Whether full fusion saves or wastes energy depends critically on the balance between two opposing effects: the elimination of DRAM traffic for intermediate activations and the increased per-access cost of the enlarged on-chip memories required to support fusion. At 160 pJ / byte for DRAM (Section 7.1.1), the outcome is workload-dependent and architecture-dependent.

On **DepFiN**, full fusion is the most energy-efficient mode for the two activation-dominant workloads. FSRCNN in single-layer mode consumes 4.38× the full-fusion energy, and MC-CNN consumes 1.73×. The explanation is that these workloads generate massive intermediate-activation volumes (Section 7.3.1), and the DRAM energy eliminated by fusion vastly exceeds the extra on-chip cost (576 KB for FSRCNN, 522 KB for MC-CNN). Partial fusion also benefits, sitting at 1.47× full for FSRCNN and 1.43× for MC-CNN.

For the weight-dominant workloads on DepFiN, the picture reverses. VGG16 single-layer execution costs only 7.4% of the full-fusion energy, and ResNet18 costs 17.3%. Fusion requires a weight memory of 14 366 KB (VGG16) or 10 738 KB (ResNet18) to hold all weights on chip simultaneously. Every weight access through these large memories incurs an elevated per-access energy that compounds across billions of multiply accumulations (MACs). Because the intermediate-activation volume is comparatively small, the DRAM savings are insufficient to compensate. Partial fusion partially mitigates this: its normalized energy is 0.49 for VGG16 and 0.75 for ResNet18, confirming that fusing only a few layers at a time requires less inflated memory.

On **Eyeriss**, full fusion is roughly energy-neutral for FSRCNN (single = 0.985× full),

suggesting that the DRAM savings and the register overhead nearly cancel. For all other workloads, full fusion is again the most expensive mode: MC-CNN single costs 45.0% of full, VGG16 single costs 5.9%, and ResNet18 single costs 12.6%. The Eyeriss energy penalty is driven by the replicated register architecture: with 16 384 PEs for VGG16, the aggregate weight register capacity is 19.7 million bytes ($1{,}200 \times 16{,}384$), each contributing to per-access energy. Despite the high DRAM cost, the vast volume of on-chip accesses through these large registers dominates the energy budget. Partial fusion is consistently cheaper than full fusion on Eyeriss, with normalized values of 0.66 (FSRCNN), 0.65 (MC-CNN), 0.39 (VGG16), and 0.69 (ResNet18).

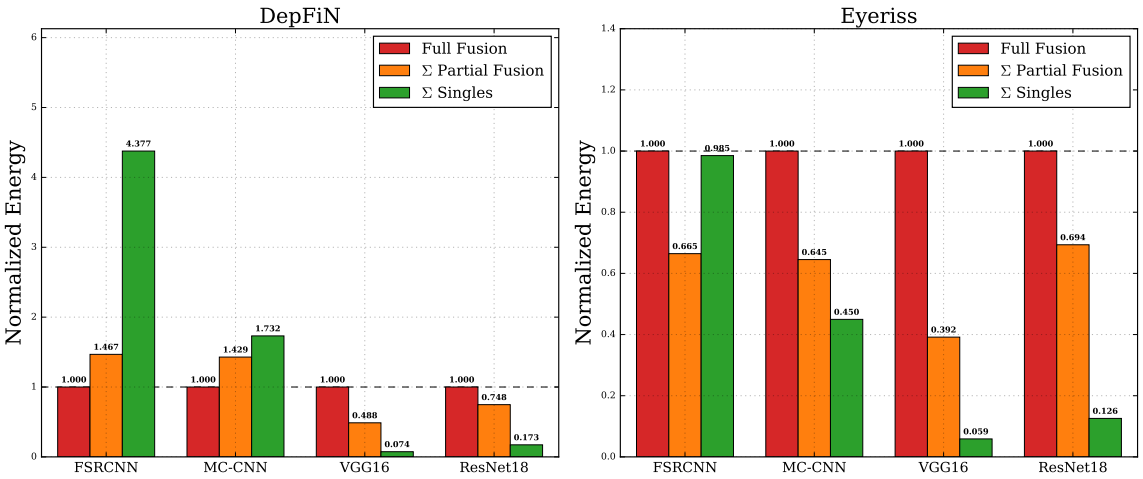

Figure 7.7: Scenario A: Normalized energy (Full Fusion = 1.0). Bars below 1.0 indicate lower energy than full fusion. Values annotated above each bar.

Latency (Figure 7.8). Full fusion delivers latency benefits whose magnitude depends on the workload and architecture.

For the activation-dominant workloads, full fusion is clearly faster on both architectures. FSRCNN enjoys a 47.6% latency saving on DepFiN and 45.6% on Eyeriss, while MC-CNN saves 42.3% on DepFiN and 15.7% on Eyeriss (Figure 7.9). These workloads have large intermediate-activation volumes relative to their weight volumes, so eliminating the DRAM round-trips for activations removes the dominant bottleneck from the execution timeline.

For the weight-dominant workloads the behavior diverges across architectures. On Eyeriss, full fusion still reduces latency compared to the single-layer sum: VGG16 saves 21.2% and ResNet18 saves 7.8%. On DepFiN, however, full fusion is *slower* than the single-layer sum: VGG16 loses 11.3% and ResNet18 loses 30.3%. Eyeriss avoids this penalty because its distributed per-PE register architecture keeps each layer's weight factors local, allowing to

compute without a unique shared memory contention point for all the layers, the weight memory.

Partial fusion tracks the full-fusion latency closely in most cases: the ratio ranges from 0.87× (ResNet18, DepFiN) to 1.07× (VGG16, Eyeriss), indicating that the latency benefit of fusion is captured almost entirely by pairwise fusion, with little additional gain from fusing all layers simultaneously.

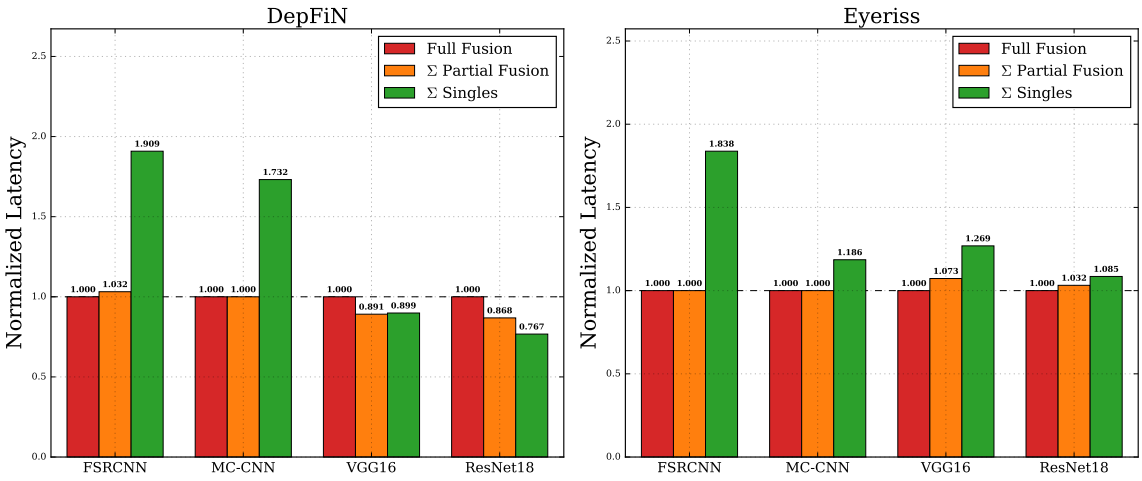

Figure 7.8: Scenario A: Normalized latency (Full Fusion = 1.0). Bars above 1.0 indicate longer latency than full fusion.

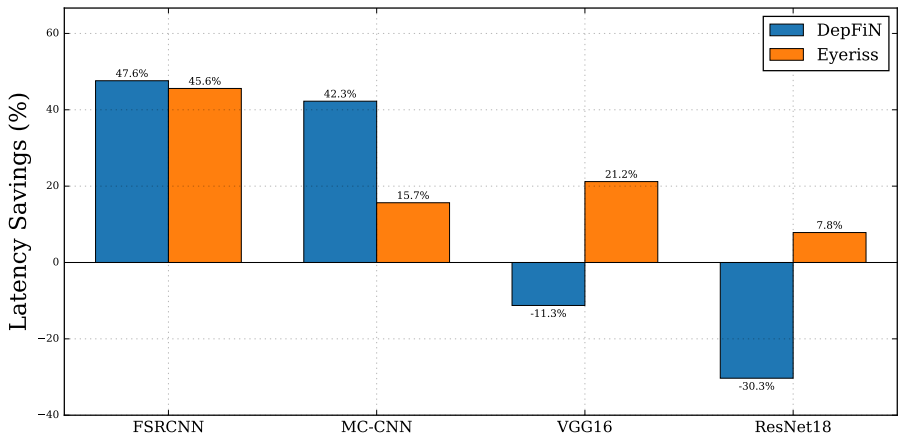

Figure 7.9: Scenario A: Latency savings of full fusion relative to single-layer execution. Positive values indicate fusion is faster; negative values indicate fusion is slower.

Energy-Delay Product (Figure 7.10). The EDP combines energy and latency into a single efficiency metric.

On **DepFiN**, full fusion is the clear EDP winner for the activation-dominant workloads. FSRCNN's single-layer EDP is 6.44× the full-fusion value, and MC-CNN is 2.31×: both energy and latency favor fusion, so EDP amplifies the advantage. For the weight-dominant workloads, however, the energy penalty of the weight memory dominates. VGG16 single-layer EDP is 0.035× full and ResNet18 is 0.073× full. Partial fusion occupies a middle ground: it captures most of the latency benefit while avoiding the worst of the memory-inflation energy cost, yielding normalized EDPs of 1.17 (FSRCNN), 1.10 (MC-CNN), 0.23 (VGG16), and 0.36 (ResNet18).

On **Eyeriss**, the distributed register overhead shifts the balance decisively against full fusion. Not for FSRCNN, whose 45.6% latency saving is comparable to DepFiN's, has a full-fusion energy roughly equal to the single-layer value (1.02×), resulting in a single-layer EDP of 1.75× the full-fusion value, meaning full fusion is still more efficient on EDP. For MC-CNN, single-layer EDP is 0.52× full, and for VGG16 and ResNet18 it drops to 0.074× and 0.13×, respectively. Partial fusion outperforms full fusion on EDP for every Eyeriss workload, with values ranging from 0.41 (VGG16) to 0.70 (ResNet18).

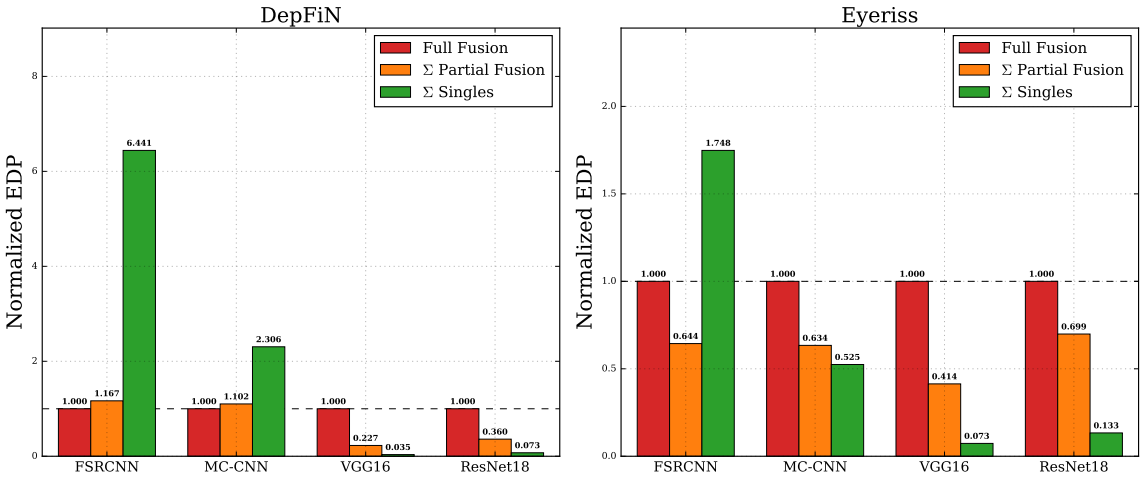

Figure 7.10: Scenario A: Normalized EDP (Full Fusion = 1.0). Bars below 1.0 indicate better efficiency than full fusion.

DRAM traffic reduction. The fundamental premise of layer fusion is the elimination of intermediate DRAM transfers, and the data confirm that this mechanism works as expected. Full fusion reduces total DRAM traffic (reads + writes) by 94.8% for FSRCNN, 85.3% for MC-CNN, 59.9% for VGG16, and 25.1% for ResNet18, relative to single-layer execution. These numbers are identical across the two architectures because the DRAM access pattern depends on the tiling of the mapping, and we are trying the same for the tile size for both the architecture for fairness of evaluation.

The reduction is largest for the activation-dominant workloads because their intermediate activations comprise the overwhelming majority of total DRAM traffic in single-layer mode. For the weight-dominant workloads, weights dominate the transfer volume, and since weights are loaded from DRAM regardless of fusion, the relative saving is smaller.

Considerations. Full fusion achieves the elimination of inter-layer DRAM traffic, delivering reductions of 25–95% depending on the workload. This translates into significant latency improvements for activation-dominant workloads (42–48% on DepFiN, 16–46% on Eyeriss).

The energy outcome depends on the interplay between the DRAM cost saved and the on-chip memory overhead incurred. At 160 pJ / byte, full fusion is energy-beneficial on DepFiN for activation-dominant workloads (FSRCNN, MC-CNN), where the eliminated DRAM traffic is large and the feature memory growth is modest. On Eyeriss, the replicated register architecture inflates the on-chip energy to the point where full fusion is energy-neutral at best (FSRCNN) and energy-adverse for all other workloads. For weight-dominant workloads (VGG16, ResNet18), full fusion is energy-adverse on both architectures because the massive weight memories required for fusion drive per-access energy far above the DRAM savings.

On EDP, full fusion wins decisively on DepFiN for the activation-dominant workloads and marginally on Eyeriss for FSRCNN; for all other cases, partial fusion or single-layer execution is preferable. The implication is that the optimal fusion strategy is workload-dependent and architecture-dependent: activation-dominant workloads on architectures with shared on-chip memories (DepFiN) benefit most from aggressive fusion, while weight-dominant workloads and architectures with replicated per-PE storage (Eyeriss) favor partial or no fusion.

7.3.2. Partial-Sized Architecture: Fusion Mode Comparison (Scenario B)

Motivation

Scenario A (Section 7.3.1) sized the architecture to support full fusion of the entire workload. The resulting memories are large, particularly on DepFiN, where VGG16 requires a 14.4 MB weight memory and ResNet18 requires 10.7 MB. Scenario B smaller architecture for each workload, is large enough to fuse any two consecutive layers, and then comparing partial-fusion execution against single-layer execution on that same hardware. Full fusion is not attempted, because the reduced memories cannot accommodate it. Table 7.9 lists

the resulting configurations.

Table 7.9: Architecture configurations used for Scenario B (partial-fusion-sized). Compared with Scenario A (Table 7.8), the DepFiN weight memory shrinks substantially for the weight-dominant workloads, and the Eyeriss WReg drops from architecture-wide values to the pair-wise minimum.

		DepFiN		Eyeriss	
Workload	PEs	fmem	wmem	GB	WReg
FSRCNN	16×128 / 128×16	248 KB	9 KB	128 KB	384
MC-CNN	8×256 / 256×8	396 KB	22 KB	128 KB	384
VGG16	16×128 / 512×32	112 KB	4 610 KB	128 KB	576
ResNet18	16×128 / 512×32	48 KB	4 608 KB	128 KB	384

The memory reductions are conspicuous for the weight-dominant workloads on DepFiN. VGG16's weight memory drops from 14 366 KB (Scenario A) to 4 610 KB, a 3.1× reduction; ResNet18 drops from 10 738 KB to 4 608 KB (2.3×). On Eyeriss, VGG16's Weight Register (WReg) falls from 1 200 to 576 bytes per PE, and ResNet18's WReg from 902 to 384 bytes. The activation-dominant workloads see smaller but still meaningful reductions on DepFiN (FSRCNN Feature Memory from 576 KB to 248 KB, Weight Memory from 19 KB to 9 KB), while on Eyeriss their configuration is unchanged because the Global-Buffer and Weight Register were already set at the pairwise minimum.

Results

Figure 7.11 presents the normalized comparison (partial fusion = 1.0) for energy, latency, and EDP.

Energy. On **DepFiN**, the energy outcome mirrors the workload-type split observed in Scenario A. For the activation-dominant workloads, partial fusion is *cheaper* than single-layer execution: FSRCNN single-layer energy is 3.03× the partial-fusion value, and MC-CNN is 1.21×. At 160 pJ / byte, the DRAM traffic eliminated by fusing even two consecutive layers outweighs the per-access cost increase from the modestly sized feature memories (248 KB for FSRCNN, 396 KB for MC-CNN). Notably, the Scenario B partial-fusion energy is also lower than the Scenario A full-fusion energy in absolute terms, because the smaller memories reduce the per-access overhead.

For the weight-dominant workloads on DepFiN, single-layer execution remains substantially cheaper: VGG16 single costs only 15.6% of the partial-fusion energy, and ResNet18

costs 27.7%. Even though the weight memory has been reduced by 2–3× relative to Scenario A, 4.6 MB is still large enough to inflate per-access energy well above the DRAM savings.

On **Eyeriss**, partial fusion is more expensive than single-layer execution for every workload. FSRCNN is closest to break-even (single = 0.88× partial), while the weight-dominant workloads show larger penalties: MC-CNN single costs 40.5% of partial, VGG16 costs 12.9%, and ResNet18 costs 18.1%. The replicated register architecture amplifies the on-chip overhead, as discussed in Section 7.3.1.

Latency. The latency pattern mirrors Scenario A closely. Partial fusion saves 45.9% on FSRCNN and 42.3% on MC-CNN for DepFiN, comparable to the 47.6% and 42.3% savings of full fusion in Scenario A. On Eyeriss the activation-dominant savings are identical to Scenario A (45.6% for FSRCNN, 15.7% for MC-CNN), because the architecture configuration has not changed for these workloads.

For the weight-dominant workloads the latency picture improves relative to Scenario A. On DepFiN, VGG16 is now essentially latency-neutral (+0.8% saving), a marked improvement over the −11.3% penalty of full fusion in Scenario A, because the smaller weight memory reduces the scheduling overhead. ResNet18 is still slower under fusion (−7.5%), but the penalty is less severe than the −30.3% in Scenario A. On Eyeriss, VGG16 and ResNet18 retain the same modest savings (15.5% and 4.9%) observed in Scenario A.

EDP. On **DepFiN**, partial fusion wins on EDP for both activation-dominant workloads. FSRCNN's single-layer EDP is 5.61× the partial-fusion value, and MC-CNN is 2.08×: both energy and latency favour fusion, and EDP amplifies the advantage. For the weight-dominant workloads, single-layer execution dominates: VGG16 single-layer EDP is 0.157× partial, and ResNet18 is 0.257× partial.

On **Eyeriss**, partial fusion wins on EDP for FSRCNN (single = 1.62× partial), where the 45.6% latency saving outweighs the modest 1.13× energy penalty. For all other Eyeriss workloads, single-layer execution wins: MC-CNN single EDP is 0.48× partial, VGG16 is 0.15×, and ResNet18 is 0.19×.

In total, partial fusion wins on EDP in three of the eight architecture–workload combinations (FSRCNN and MC-CNN on DepFiN, FSRCNN on Eyeriss), compared with the equivalent three of eight in Scenario A.

Comparison with Scenario A. Scenario B demonstrates that sizing the architecture for pairwise fusion rather than full fusion brings two clear advantages.

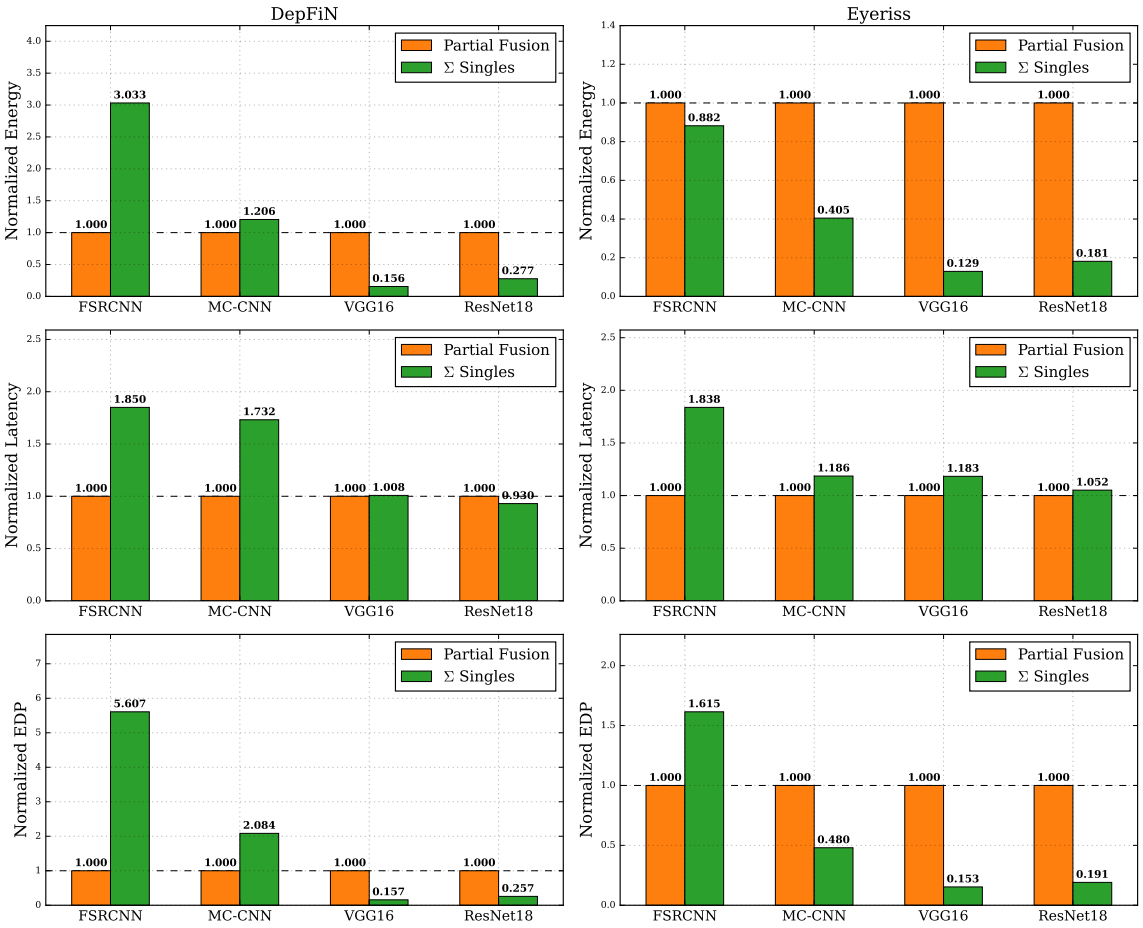

Figure 7.11: Scenario B: Normalized energy (top), latency (middle), and EDP (bottom), with partial fusion = 1.0. Bars below 1.0 indicate the single-layer sum is more efficient.

The first is the substantial memory reduction (Table 7.9), which lowers energy cost access for the memories and registers. The second is that the latency benefit of fusion is almost fully preserved: partial fusion captures 96–100% of the full-fusion latency saving for the activation-dominant workloads, and it eliminates or greatly reduces the worst-case latency penalty that full fusion imposed on DepFiN ($-11.3\% \rightarrow +0.8\%$ for VGG16; $-30.3\% \rightarrow -7.5\%$ for ResNet18).

The EDP verdict is identical in scope: three of eight architecture–workload combinations favor fusion in both scenarios (FSRCNN and MC-CNN on DepFiN, FSRCNN on Eyeriss). The weight-dominant workloads continue to favor single-layer execution on both architectures, regardless of whether the design targets full or partial fusion.

Considerations A partial-fusion-sized architecture delivers nearly the same latency and EDP benefit as a full-fusion architecture, at a fraction of the memory cost. For activation-dominant workloads on DepFiN, partial fusion wins on energy, latency, and EDP simultaneously, making it the strictly dominant strategy. On Eyeriss, partial fusion wins on EDP only for the most activation-heavy workload (FSRCNN), because the replicated register overhead partially offsets the DRAM savings. For weight-dominant workloads, the on-chip memory overhead of fusion still outweighs the DRAM savings, and single-layer execution remains the more efficient choice on both architectures.

7.3.3. Energy Ratio vs. Register Size (Eyeriss)

Motivation

Sections 7.3.1 and 7.3.2 compared fusion modes at a single architecture point per workload. This case study will quantify the energy penalty of fusion in three different configuration of PE of Eyeriss, each requiring progressively larger registers.

Two weight-dominant workloads, ResNet18 and VGG16, are examined because they span a range of WReg values wide enough to reveal the scaling trend. For each workload, three PE counts are tested (16 384, 8 192, 4 096), each paired with the minimum WReg determined by CS1. The activation-dominant workloads (FSRCNN, MC-CNN) are excluded from the sweep because they share a single WReg = 384 regardless of PE count, so there is no register-size axis to sweep.

Results

ResNet18 (Figure 7.12). Table 7.10 reports the three configurations. At the largest array (512×32, 16 384 PEs, WReg = 902) the full-to-single energy ratio is 8.0×. Halving the PE count to 8 192 and doubling WReg to 1 800 raises the ratio to 11.7×, a 1.5× increase. At 4 096 PEs with WReg = 3 600, the ratio reaches 19.6×.

Table 7.10: Eyeriss fixed-config scaling for ResNet18 (Scenario A). WReg is the minimum feasible value from CS1. Energy ratio = Full / Single.

Config	PEs	WReg	Full (μJ)	Single (μJ)	Ratio
512×32	16 384	902	2.37×10^4	2.98×10^3	8.0×
256×32	8 192	1 800	3.25×10^4	2.78×10^3	11.7×
256×16	4 096	3 600	5.20×10^4	2.65×10^3	19.6×

The growth arises because two effects stack in the ratio. First, the fused energy increases with WReg because every register access touches a larger SRAM structure, and the aggregate register capacity across all PEs is roughly constant ($902 \times 16{,}384 \approx 14.8$ M bytes at the top, $3{,}600 \times 4{,}096 \approx 14.7$ M at the bottom), so the per-access cost rises while the total capacity stays flat. Second, the single-layer energy *decreases* slightly with fewer PEs because single-layer mode does not require the oversized WReg. However, the single-layer shrinkage is modest (only 5–13% per step) because the DRAM component of single-layer energy is substantial and PE-count-independent, leaving the register-driven reduction as a minor effect. The ratio growth is therefore driven primarily by the fused-energy increase (1.4–1.7× per step).

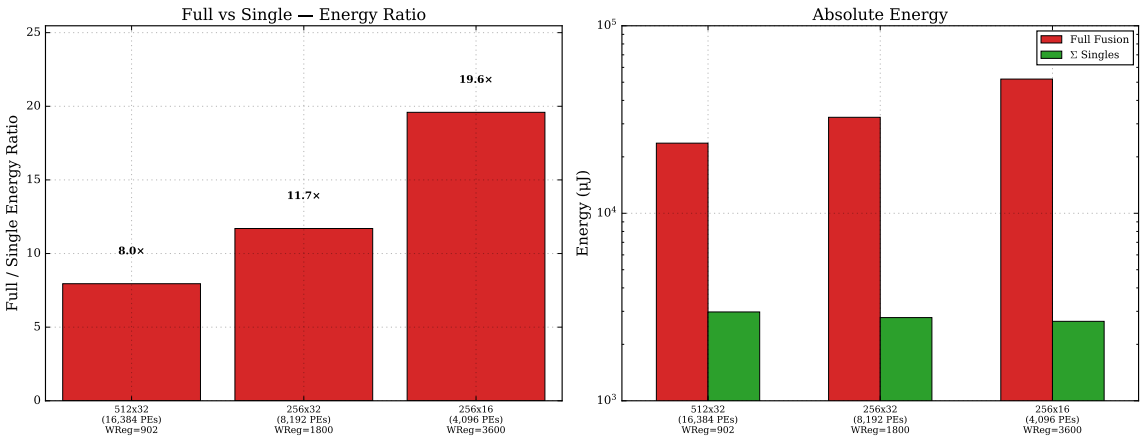

Figure 7.12: Eyeriss fixed-config scaling for ResNet18. Left: full-to-single energy ratio. Right: absolute energy on a log scale.

VGG16 (Figure 7.13). VGG16 exhibits the same pattern, amplified by its larger register requirements. At WReg = 1 200 (16 384 PEs) the ratio is 17.0×; at WReg = 2 400 (8 192 PEs) it rises to 28.9×; and at WReg = 4 800 (4 096 PEs) it reaches 54.1×. Each doubling step grows the ratio by 1.7–1.9×, consistent with the ResNet18 trend.

Table 7.11: Eyeriss fixed-config scaling for VGG16 (Scenario A).

Config	PEs	WReg	Full (μJ)	Single (μJ)	Ratio
512×32	16 384	1 200	1.63×10^5	9.59×10^3	17.0×
256×32	8 192	2 400	2.45×10^5	8.49×10^3	28.9×
256×16	4 096	4 800	4.19×10^5	7.74×10^3	54.1×

VGG16 also reveals a difference between full and partial fusion. The full-to-partial energy ratio is 2.55× at WReg = 1 200 and decreases to 1.41× at WReg = 4 800, indicating that as registers grow, partial fusion captures most of the fused-mode overhead and the gap between the two modes narrows. For ResNet18 the full-to-partial ratio ranges from 1.44× at the smallest WReg down to 1.02× at the largest, showing near-equivalent energy for full and partial fusion at large registers.

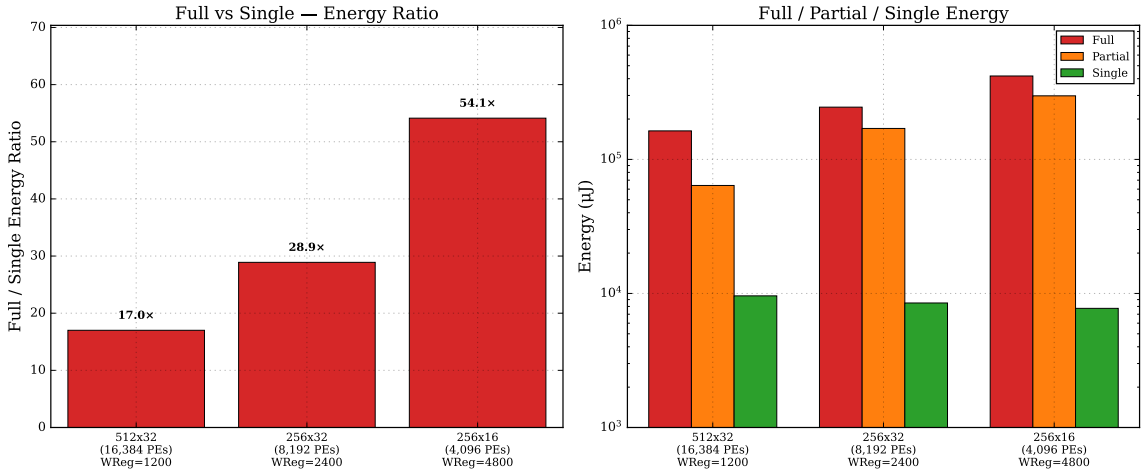

Figure 7.13: Eyeriss fixed-config scaling for VGG16. Left: full-to-single energy ratio. Right: absolute energy (full, partial, single) on a log scale.

Activation-dominant workloads. FSRCNN and MC-CNN do not participate in this sweep because their WReg remains at 384 bytes regardless of the PE configuration. At that single operating point, the full-to-single energy ratios are 1.0× (FSRCNN) and 2.2× (MC-CNN), which sit well below the weight-dominant ratios. For FSRCNN, fusion incurs essentially no energy overhead: almost all energy is spent in DRAM transfers that are common to both modes. The constant WReg reflects the fact that, for shallow networks

with small filter counts, the per-PE weight footprint is already negligible at the minimum PE count.

Scaling rule. Across both workloads, each 2× increase in WReg raises the fusion-to-single energy ratio by a factor of roughly 1.5–1.9×. The fused energy grows by 1.4–1.7× per WReg step because each register access touches a larger SRAM structure. The single-layer energy shrinks only modestly, by 1.05–1.13×, because the DRAM component of single-layer energy is substantial and PE-count-independent, so the register-driven shrinkage is a minor effect. The product of these two factors yields the observed ∼1.5–1.9× ratio growth per step.

Considerations The energy penalty of fusion on the Eyeriss architecture is not a fixed constant; it *scales super-linearly* with register size. A designer who chooses a smaller PE array to save area will face a correspondingly larger WReg and, consequently, a worse fusion-to-single energy ratio. At the extreme tested point (VGG16, WReg = 4 800), fusion consumes 54× the energy of single-layer execution. This scaling behavior reinforces the conclusion from Sections 7.3.1 and 7.3.2: fusion's energy cost is the dominant factor in the EDP trade-off, and it grows worse as the architecture is scaled down.

8 | Conclusions

8.1. Summary

This thesis explored the complex design space of multi-layer fusion on spatial DNN accelerators. To navigate and evaluate this space, the FactorFlow analytical framework was extended to model cascading inter-layer dependencies that govern fused execution. The research encompassed the theoretical modeling, implementation of the model, and validation using an hardware architecture specialized for fusion [10], followed by an experimental evaluation. To quantify the inherent performance and energy trade-offs, the evaluation systematically analyzed modified DepFiN-Like and Eyeriss-Like spatial architectures across both activation-dominant and weight-dominant CNN workloads (FSRCNN, MC-CNN, VGG16, ResNet18). The study compared three execution strategies (full fusion, partial fusion, and standard single-layer execution) across two distinct hardware-sizing scenarios (Scenario A and Scenario B). This section recapitulates the key findings along two primary axes: the empirical trade-offs of layer fusion as an execution strategy, and the theoretical contribution of the formalized analytical model in making this design space navigable.

8.1.1. Formalizing and Exploring the Fused Design Space

A primary contribution of this thesis is the formalization of the complex design space inherent to multi-layer fused scheduling. The transition from single-layer to fused execution is not a mere linear concatenation of operations, but a proliferation of loop dimensions that fundamentally alters the mapping problem.

Defining Dimension Proliferation and Residency. This work accurately models the on-chip residency of data during fusion by establishing five distinct operand categories: inputs, weights, intermediate outputs, intermediate inputs, and final outputs. This precise categorization is essential for formalizing the memory-level bypass rules, effectively governing whether intermediate activations are buffered for inter-layer reuse or evicted.

Managing Cascading Dependencies. Fusion introduces strict cascading dependencies between layers. The choice of an outermost output tile dictates the required intermediate tile sizes of all preceding layers due to the accumulation of filter affine dimension. The framework translates these dataflow restrictions into exact mathematical constraints, allowing for a precise characterization of the reuse-recompute trade-off.

Sizing and pruning the new Search Space. Unconstrained, the compounding permutations of a fused loop nest cause the map-space to explode to intractable scales, exceeding 10^{515} possible configurations for a 11-layer architecture. Unguided exploration of such a space has effectively zero probability of discovering valid, high-quality mappings. By implementing rigorous constraint-based pruning, this work dynamically eliminates structurally invalid configurations that violate hardware dataflow, spatial factorization, or memory capacity limits.

8.1.2. Impact of Layer Fusion

The central premise of layer fusion is that eliminating the intermediate-activation round-trips through external memory can reduce both energy and latency. The experiments confirm part of this premise and, equally importantly, reveal its boundaries: the outcome depends on the workload's activation-to-weight traffic ratio, the architecture's memory organization, and the per-access energy cost of DRAM.

DRAM traffic reduction. Full fusion achieves its objective of reducing DRAM traffic, it is reduced by 25–95% depending on the workload (94.8% for FSRCNN, 85.3% for MC-CNN, 59.9% for VGG16, 25.1% for ResNet18). Activation-dominant workloads benefit the most because their intermediate activations constitute the overwhelming majority of off-chip traffic in single-layer mode. These reductions are identical across the two architectures because the DRAM access pattern is determined by the tiling of the mapping, and we use the same tile sizes on both architectures for fairness of evaluation.

Latency benefit. The DRAM-traffic reduction translates into substantial latency improvements for activation-dominant workloads: 42–48% on DepFiN and 16–46% on Eyeriss. For weight-dominant workloads the results depend on the architecture. On Eyeriss, full fusion still reduces latency (7.8–21.2%), because the distributed per-PE register architecture avoids a single shared-memory contention point for all layers' weights. On DepFiN, however, full fusion is *slower* than single-layer execution (up to 30.3% latency increase for ResNet18) because the very large shared weight memory (10.7–14.4 MB) creates a bandwidth bottleneck that outweighs the DRAM savings. Partial (pairwise) fusion captures

87–103% of the full-fusion latency benefit, indicating that nearly all the inter-layer traffic saving is realized by fusing just two consecutive layers at a time.

Energy: a workload-dependent and architecture-dependent outcome. Unlike the simpler narrative of "fusion always increases energy," the outcome depends critically on the interplay between the DRAM energy saved and the on-chip memory overhead incurred, and consequently on the DRAM per-access energy cost assumed in the model. At the 160 pJ / byte DRAM cost used in this work (Section 7.1.1), the results split along workload type and architecture:

- On **DepFiN**, full fusion is energy-beneficial for activation-dominant workloads: FSRCNN single-layer energy is 4.38× the full-fusion value, and MC-CNN is 1.73×. The eliminated DRAM traffic is so massive that the on-chip overhead is outweighed by the DRAM savings.
- For weight-dominant workloads on DepFiN, the picture reverses: VGG16 single-layer energy is only 7.4% of the full-fusion value, and ResNet18 is 17.3%. Fusion requires weight memories of 14.4 MB (VGG16) or 10.7 MB (ResNet18), whose per-access energy compounds across billions of MACs, far exceeding the DRAM savings.
- On **Eyeriss**, full fusion is roughly energy-neutral for FSRCNN (0.985× full) and energy-adverse for all other workloads (MC-CNN single costs 45% of full, VGG16 costs 5.9%, ResNet18 costs 12.6%). The replicated per-PE register architecture amplifies the on-chip energy: with 16 384 PEs for VGG16, the aggregate weight register capacity is 19.7,M bytes (1,200 × 16,384), each register access contributing to per-access energy.

A lower DRAM energy cost would reduce the benefit of eliminating DRAM traffic and shift the balance further against fusion; a higher DRAM cost would extend fusion's energy advantage to more workload–architecture combinations. The success of fusion is therefore sensitive to the DRAM

Energy–Delay Product. When energy and latency are combined into the EDP metric, full fusion wins in three of the eight architecture–workload combinations: FSRCNN on DepFiN (6.44× single/full), MC-CNN on DepFiN (2.31×), and FSRCNN on Eyeriss (1.75×). In all three, the activation-dominant nature of the workload ensures that both energy and latency favor fusion, amplifying the EDP advantage. For the remaining five combinations, single-layer execution is the more efficient choice, with the weight-dominant workloads showing particularly unfavorable ratios (VGG16 on Eyeriss: 0.074×; ResNet18

on DepFiN: 0.073×).

This result clarifies the role of layer fusion: *fusion is primarily a latency optimization whose energy trade-off is workload-dependent.* Its benefit grows with the activation-to-weight traffic ratio and diminishes as the on-chip memory overhead increases. Whether the trade-off is acceptable depends on the design priority, the workload class, and the external memory technology: energy-constrained systems or weight-dominant workloads should favor single-layer execution.

Partial fusion as a practical compromise. Partial fusion consistently captures most of the latency benefit while reducing the memory and energy cost. In Scenario B, sizing the architecture for pairwise rather than full fusion reduced the weight memory size up to 3.1× on DepFiN (VGG16: 14 366 KB→4 610 KB; ResNet18: 10 738 KB→4 608 KB) and the per-PE weight register (WReg) size on Eyeriss (VGG16: 1 200→576; ResNet18: 902→384). The latency benefit was almost fully preserved: partial fusion captured 96–100% of the full-fusion latency saving for activation-dominant workloads, and it eliminated or greatly reduced the worst-case latency penalty on DepFiN (−11.3% → +0.8% for VGG16; −30.3% → −7.5% for ResNet18). The EDP verdict was identical in scope: three of eight architecture–workload combinations favored fusion in both scenarios. For activation-dominant workloads on DepFiN, partial fusion won on energy, latency, and EDP simultaneously, making it the strictly dominant strategy.

Super-linear scaling of the energy penalty with PE configurations. The experiment on Eyeriss (Section 7.3.3) revealed that the fusion-to-single energy ratio with different PE configuration: it *scales super-linearly* with register size. Each 2× increase in per-PE WReg raised the ratio by 1.5–1.9× (ResNet18: 8.0× → 11.7× → 19.6×; VGG16: 17.0× → 28.9× → 54.1×). The growth arises because the fused energy increases with WReg (each access touches a larger SRAM structure) while the single-layer energy decreases only modestly (the DRAM component is PE-count-independent). Shrinking the PE array to save area will face a proportionally larger WReg and, consequently, a worse fusion-to-single energy ratio.

8.2. Future Work

While the framework developed in this thesis demonstrates the feasibility of analytical fused-layer mapping evaluation, remaining limitations define clear directions for future research.

8.2.1. Automated Mapper for Fused Workloads

The most impactful limitation in the state of the art is the absence of an automated mapper for multi-layer fused workloads. Currently, fused mappings must be specified manually through the architecture definition file, restricting exploration to expert-guided configurations.

While the constraint-based pruning developed in this work successfully eliminates physically impossible hardware mappings, the remaining valid map-space is still far too large for exhaustive or simple heuristic searches (such as standard linear or quadratic methods). Extending the existing mapper infrastructure to automatically explore the fused map-space will require the development of aggressive, domain-specific heuristics. These new heuristics must deliberately reduce the search space to achieve computational tractability, which inherently comes at the cost of potentially losing absolute optimal solutions. Designing search algorithms that can intelligently navigate this trade-off, sacrificing guaranteed optimality to reliably find high-efficiency configurations within a reasonable timeframe, is the critical next step to transform this tool into a fully autonomous design-space exploration engine.

8.2.2. Inter-Layer Pipeline Parallelism

The current analytical model enforces strictly sequential, depth-first layer execution: processing tiles sequentially. However, architectures can also exploit *inter-layer pipeline parallelism*, where different layers execute concurrently on spatially partitioned, disjoint subsets of the PE array. Extending the framework to support this paradigm would require formulating a new latency model capable of capturing overlapping execution intervals, pipeline stalls, and inter-partition bandwidth constraints. This extension will broad the class of fused-layer schedules that can be analytically modeled and compared against depth-first approaches.

8.2.3. Fine-Grained Modeling of Innermost Dimension-Sum Reuse

When two loop dimensions contribute to the same operand index through an affine sum (e.g., $y + r$ in a standard convolution), the current cost model conservatively assumes independent memory accesses. It does not completely capture the data overlap between consecutive iterations of the sliding window within a single spatial tile. This simplification leads to a safe, but over-estimated, count of Memory Operations (MOPs) at the innermost scratchpad levels. Implementing a more fine-grained footprint analysis to account for this

sliding-window overlap would yield highly accurate energy estimates. This refinement is particularly important for evaluating workloads featuring large filter dimensions.

8.2.4. Computational Scaling via Compiled Systems Languages

While the current Python-based implementation of our work is highly effective for mathematical prototyping, architectural modeling, and validation, it incurs inherent interpreter overhead. Furthermore, it is constrained by the Global Interpreter Lock (GIL), which precludes true shared-memory multithreading. Transitioning the analytical engine to a compiled, systems-level language (such as C++ or Rust) is a crucial next step. Eliminating these computational bottlenecks and enabling highly parallel, lightweight multithreading is practically mandatory to support the immense number of cost-model evaluations required by the automated heuristic mappers proposed in Section 8.2.1.

8.2.5. Joint Optimization with Graph-Level Schedulers

This thesis operates strictly at the *dataflow level*, exploring the spatial and temporal mapping of a predetermined set of fused layers. A highly promising complementary direction involves *graph-level* scheduling, determining mathematically *which* layers to fuse and how to optimally partition the neural network graph into distinct fusion segments. Tools such as Optimus address this macro-level partitioning problem. Integrating a graph-level topology scheduler with the fine-grained, constraint-aware analytical dataflow model developed in this work would enable the joint optimization of both the fusion graph and the hardware mapping.

Bibliography

[1] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. High-performance tensor contractions for gpus. *Procedia Computer Science*, 80:108–118, 2016. URL `https://doi.org/10.1016/j.procs.2016.05.302`. International Conference on Computational Science 2016, ICCS 2016, 6–8 June 2016, San Diego, California, USA.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[3] Apollodoro. *Biblioteca, I, 9, 26, trad. it. M.G. Ciani*. Mondadori, 1996.

[4] Y. Bengio and Y. LeCun, editors. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `https://iclr.cc/archive/www/doku.php%3Fid=iclr2015:accepted-main.html`.

[5] X. Cai, Y. Wang, and L. Zhang. Optimus: towards optimal layer-fusion on deep learning processors. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 67–79, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384728. doi: 10.1145/3461648.3463848. URL `https://doi.org/10.1145/3461648.3463848`.

[6] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017. doi: 10.1109/JSSC.2016.2616357.

[7] C. Dong, C. C. Loy, and X. Tang. Accelerating the super-resolution convolutional neural network. *CoRR*, abs/1608.00367, 2016. URL `http://arxiv.org/abs/1608.00367`.

[8] A. Einstein. The foundation of the general theory of relativity. *Annalen der Physik*, 354(7):769–822, 1916.

[9] M. Gilbert, Y. N. Wu, J. S. Emer, and V. Sze. Looptree: Exploring the fused-layer dataflow accelerator design space. arXiv preprint, 2024.

[10] K. Goetschalckx, F. Wu, and M. Verhelst. Depfin: A 12-nm depth-first, high-resolution cnn processor for io-efficient inference. *IEEE Journal of Solid-State Circuits*, 58(5):1425–1435, 2023.

[11] K. He, X. Zhang, S. Ren, and S. Jian. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] JEDEC Committee JC-42.6. *Low Power Double Data Rate 4 (LPDDR4) JESD209-4D*. JEDEC Solid State Technology Association. URL `http://www.jedec.org/standards-documents/results/jesd209-4`. Accessed: Oct. 9, 2022.

[13] 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*, pages 1–12, 2017.

[14] S.-C. Kao and T. Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[15] 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 (MICRO '52)*, pages 754–768, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358252. URL `https://doi.org/10.1145/3352460.3358252`.

[16] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[17] W. Luo, A. G. Schwing, and R. Urtasun. Efficient deep learning for stereo matching. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5695–5703, 2016. doi: 10.1109/CVPR.2016.614.

[18] Z. Mandi, H. Bharadhwaj, V. Moens, S. Song, A. Rajeswaran, and V. Kumar. Cacti: A framework for scalable multi-task multi-scene visual imitation learning, 2023. URL `https://arxiv.org/abs/2212.05711`.

[19] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst. Zigzag: Enlarging joint

architecture-mapping design space exploration for dnn accelerators. *IEEE Transactions on Computers*, 70(8):1160–1174, 2021. doi: 10.1109/TC.2021.3059962.

[20] L. Mei, K. Goetschalckx, A. Symons, and M. Verhelst. Defines: Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583, Los Alamitos, CA, USA, Mar. 2023. IEEE Computer Society.

[21] A. Parashar, P. Raina, Y. S. Shao, Y.-H. 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 (ISPASS)*, pages 304–315, 2019. doi: 10.1109/ISPASS.2019.00042.

[22] M. Ronzani and C. Silvano. Factorflow: Mapping gemms on spatial architectures through adaptive programming and greedy optimization. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, ASPDAC '25, page 706–712, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400706356. doi: 10.1145/3658617.3697670. URL `https://doi.org/10.1145/3658617.3697670`.

[23] M. Ronzani and C. Silvano. Quickflow: An efficient local search method to map convolutions on spatial architectures. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2025. doi: 10.1109/ICCAD66269.2025.11240877.

[24] M. Ronzani and C. Silvano. Quickflow: An efficient local search method to map convolutions on spatial architectures, 2025.

[25] F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65(6):386–408, 1958.

[26] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[27] Y. Tian. An analytical formula of population gradient for two-layered relu network and its applications in convergence and critical point analysis. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 3404–3413. JMLR.org, 2017.

[28] E. Valpreda, P. Morì, N. Fasfous, M. R. Vemparala, A. Frickenstein, L. Frickenstein, W. Stechele, C. Passerone, G. Masera, and M. Martina. Hw-flow-fusion: Inter-layer scheduling for convolutional neural network accelerators with dataflow architectures. *Electronics*, 11(18):2933, 2022. doi: 10.3390/electronics11182933.

[29] L. Waeijen, S. Sioutas, M. Peemen, M. Lindwer, and H. Corporaal. ConvFusion: A model for layer fusion in convolutional neural networks. *IEEE Access*, 9:15888–15904, 2021.

[30] Y. N. Wu, J. S. Emer, and V. Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019. doi: 10.1109/ICCAD45719.2019.8942149.

List of Figures

List of Tables

List of Acronyms

Acronym	Description
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DF	Depth-First
DL	Deep Learning
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
EDP	Energy-Delay Product
EEN	Extended Einsum Notation
FMEM	Feature Memory
GB	Global Buffer
GEMM	General Matrix Multiplication
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
InReg	Input Register
IntReg	Intermediate Register
IS	Input-Stationary
LBL	Layer-By-Layer
MAC	Multiply-and-Accumulate
MOP	Memory Operation
MP	McCulloch Pitts
OS	Output-Stationary
OutReg	Output Register
PE	Processing Element
ReLU	Rectified Linear Unit
SA	Spatial Architecture

Acronym	Description (continued)
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
SoC	System on Chip
SRAM	Static Random-Access Memory
WMEM	Weight Memory
WReg	Weight Register
WS	Weight-Stationary

List of Symbols

Symbol	Description
A_i	Intermediate activation tensor between layer i and $i+1$
a	Neuron's activation
B_l	Physical read or write bandwidth at level l
b	Neuron's bias
C	Input channels / filter depth
$\mathcal{D}_i$	Set of dimensions relevant to layer i
D	Set of normal kernel dimensions
Δ_l	Stall cycles at level l
E	Total energy consumption
EDP	Energy-Delay Product
e^{MAC}	Energy of a single multiply-accumulate operation
e_l^{rd}, e_l^{wr}	Per-scalar read and write energies at level l
F	Number of fused layers
f_k	Tiling factors assigned to level k
I / $Input$	Input activation tensor
$\mathcal{L}_i$	Layer i
L / Λ	Total execution latency
L_m	Number of memory levels
M	Output channels / number of filter groups
$\mathcal{MS}$	Map-space (set of all valid mappings)
N	Batch dimension
O / $Output$	Output activation tensor
P	Output spatial height
Q	Output spatial width
R	Filter spatial height

Symbol	Description (continued)
S	Filter spatial width
$S_{available}$	Physically available on-chip memory capacity
$S_{required}$	Total scratchpad capacity required for a given reuse level
$\sigma_l^{(i)}$	Scaling factor of temporal and spatial iterations
t_d	Tile size for dimension d
$v_p(d)$	p-adic valuation of $\|d\|$
W / $Filter$	Weight tensor
X_i	Spatial width of the intermediate activation produced by layer i
Y_i	Spatial height of the intermediate activation produced by layer i
Z_i	Output channels for layer i