Seminar Report

From Art to Science: Frameworks for the Systematic Design of Tensor Accelerators

Brandon Schühlein

***Abstract*—The proliferation of machine learning (ML) into embedded systems demands specialized hardware to meet stringent power, performance, and area constraints. Domain-Specific Accelerators (DSAs) have emerged as the primary solution, yet their design presents a formidable challenge due to a vast and complex design space encompassing architecture, dataflow, and workload-specific mappings. A naive comparison between two accelerator designs can be misleading, as performance is often dominated by the quality of the mapping rather than inherent hardware capabilities. This report surveys the systematic approach to accelerator design enabled by modern modeling and mapping frameworks. We begin by examining the foundational tools, Accelergy and Timeloop, which decouple hardware specification from mapping search to enable fair comparisons. We then explore how subsequent frameworks have expanded this foundation along several axes: enlarging the mapspace with techniques like imperfect factorization (Ruby), inter-layer fusion (LoopTree), and physical data layout co-optimization (Layoutloop); and extending modeling capabilities to advanced paradigms such as sparse (Sparseloop) and Compute-in-Memory (CiMLoop) accelerators. By tracing this evolution, we highlight how these tools are transforming accelerator design from an intuition driven art into a systematic, data driven engineering discipline, paving the way for fully automated hardware-software co-design.**

I. Introduction

The last decade has witnessed the rapid integration of machine learning (ML), particularly deep neural networks (DNNs), into numerous applications. This trend is increasingly extending to the edge, with the rise of embedded ML enabling intelligent features in everyday devices, from smartphones and wearables to autonomous vehicles and smart home assistants. Executing ML workloads directly on these devices offers various advantages, including low latency, enhanced user privacy, and robust operation without reliance on constant network connectivity.

However, deploying ML models on resource-constrained embedded systems introduces a unique set of hardware demands. These devices operate under tight power and energy budgets to ensure long battery life, have a minimal physical footprint (area), and must provide low-latency inference while remaining cost-effective. Meeting these conflicting requirements with general-purpose hardware is no longer feasible. With Moore's Law slowing and Dennard scaling ending, generational CPU improvements no longer suffice for necessary performance and energy efficiency gains [1].

The solution lies in *Domain-Specific Accelerators (DSAs)*, which are custom-designed hardware tailored to the computational patterns of ML workloads. However, the design of these accelerators is itself a monumental challenge due to a vast design space. Key architectural decisions include the choice of computational paradigm (e.g., systolic arrays, Compute-in-Memory), the memory hierarchy, and the topology of the Network-on-Chip (NoC). Furthermore, the performance of a given accelerator is not absolute; it is highly dependent on the specific workload it executes and on the *mapping* - the strategy used to schedule computations and orchestrate data movement on the hardware. This dependency creates a comparison problem: how can we fairly evaluate two different accelerator designs? A naive comparison might merely reflect a suboptimal mapping choice for one architecture, rather than an inherent limitation of the hardware itself.

This is the critical role of modeling and mapping frameworks, which provide a systematic solution to this comparison problem. They combine fast analytical models with efficient mapping exploration to find a near-optimal mapping for any given architecture and workload, thereby enabling a fair comparison and ultimately paving the way for automated hardware search where the architecture itself becomes a searchable variable.

This report surveys the evolution of these modeling and mapping frameworks. We will trace their development from foundational tools for dense accelerators to more advanced systems capable of modeling an ever expanding design and mapping space, including emerging paradigms like sparse and Compute-in-Memory accelerators.

This report is structured as follows. Section II introduces the fundamental concepts of tensor computations, accelerator hardware, and the mapping problem. Section III details the foundational frameworks, Accelergy and Timeloop. Section IV and Section V survey tools that expand upon this foundation to broaden the mapping space and model advanced accelerator designs, respectively. Finally, Section VI discusses future research directions, and Section VII concludes the report.

II. Background

A. Tensor Operations

Computations in deep learning are typically operations on multidimensional arrays, or tensors [2]. These tensor operations, such as matrix-matrix multiplication and convolution, constitute the majority of the computational workload [3].

To illustrate the mathematical structure of these operations, consider a 1D convolution. This operation computes an output tensor of size $M \times P$ from an input tensor of size $C \times W$ and

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s in range(S):
  for q in range(Q):
    w = q+s
    o[q] += i[w] * f[s]
```

(a) Weight Stationary (WS)

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s in range(S):
  for q in range(Q):
    w = q+s
    o[q] += i[w] * f[s]
```

(b) Output Stationary (OS)

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```

(c) WS + Parallelism

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for q1 in range(Q1):
  for s in range(S):
    for q0 in range(Q0):
      q = q1*Q0 + q0
      w = q+s
      o[q] += i[w] * f[s]
```

(d) OS + Tiling

Fig. 1: Four different loop nests for a 1D convolution [5].

a filter tensor of size $M \times C \times R$. Using standard summation notation, this is expressed as:

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

Here, the indices m and p iterate over the output channels and spatial positions of the Output tensor, respectively.

While clear for simple cases, standard summation becomes impractical for higher-dimensional operations. The *Extended Einstein Summation* (Einsum) notation addresses this by providing a more concise representation [4]. The same 1D convolution is expressed in Einsum as:

$$\text{Output}_{m,p}^{M,P} = \text{Input}_{c,p+r}^{C,W} \times \text{Filter}_{m,c,r}^{M,C,R} \tag{2}$$

In this notation, superscripts explicitly declare the tensor dimensions, such as M for output channels and P for output width. The subscripts (m, p, c, r) define the specific computation. A key feature of Einsum is its rule for implicit summation, which occurs over any index present in the right side tensors but not the output tensor. For the 1D convolution example, this summation happens over both the channel index c and the filter index r. This compact representation serves as a common language for the frameworks discussed throughout this report.

B. Loop Nests

Tensor operations, as described by the Einsum notation, translate directly into nested loops. Each loop iterates over a specific dimension of the tensors involved in the computation. The ordering of these loops can be arbitrary for many tensor operations, because the underlying multiply-and-accumulate (MAC) operations are commutative. Different loop permutations significantly impact data reuse and memory access efficiency [5], [6]. Figure 1a and Figure 1b illustrate two distinct loop orderings for a 1D convolution.

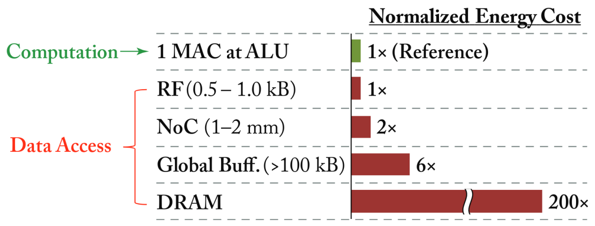


Fig. 2: Normalized energy cost relative to the computation of one MAC operation at ALU, illustrating the dominance of data access over computation [5].

Furthermore, these loops can be manipulated to further optimize execution. Large loops can be broken into smaller iterations by tiling, as shown in Figure 1d. Tiling partitions the computation into smaller chunks, allowing the tiles to fit within the limited capacity of faster on-chip memory levels. Additionally, loops can be executed in parallel using constructs like parallel-for (see Figure 1c), distributing the loop iterations across multiple *Processing Elements* (PEs) or cores within an accelerator [5]. The interplay of loop ordering, tiling, and parallelism forms the basis for defining concrete execution strategies known as mappings, which are explained in Section II-F.

C. Data Movement Bottleneck

A core challenge in accelerator design is the *data movement bottleneck*. The root of this problem is the high cost of individual data accesses. As illustrated in Figure 2, fetching a single word from off-chip DRAM can be over two orders of magnitude more costly in terms of energy than performing the computation itself; the latency disparity is often even greater [5], [7]. Consequently, the overall process of moving data between memory and compute units becomes the dominant factor in system performance, as PEs are frequently stalled awaiting data, leading to underutilization [6].

To combat this, accelerators employ multi-level memory hierarchies. The effectiveness of this strategy hinges on exploiting data reuse to minimize expensive off-chip accesses. This reuse is primarily categorized as temporal, where a single PE reuses a data element over time, and spatial, where a single data element is shared across multiple PEs [5].

Crucially, the degree of data reuse that can be achieved is determined not by hardware alone, but by the interplay between hardware and the chosen strategy for orchestrating computation and data movement.

D. Hardware

The pursuit of computational efficiency for DNN workloads has led to the development of specialized hardware, a necessary shift towards domain-specific designs as argued by Hennessy and Patterson [1]. These architectures can be broadly classified into two main paradigms: temporal and spatial [5]. Temporal architectures, like GPUs, feature numerous compute units executing instructions in parallel on data fetched from a centralized memory system. In contrast, modern custom

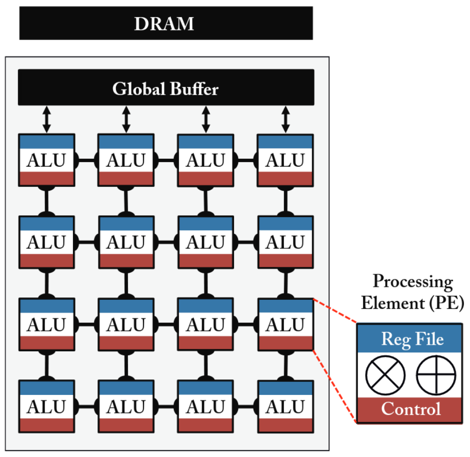


Fig. 3: A generic spatial DNN accelerator architecture, highlighting the core components: a PE array, a memory hierarchy, and a NoC [5].

DNN accelerators predominantly adopt spatial architectures, characterized by a distributed array of PEs that enables direct data flow between compute units.

As illustrated in Figure 3, a generic spatial accelerator is typically composed of three fundamental hardware building blocks organized to facilitate data reuse and parallel computation:

1) Processing Elements (PEs): At the core of the design is an array of PEs, each typically containing a MAC unit and a small local memory for holding operands stationary to exploit temporal reuse.
2) Memory Hierarchy: A multi-level memory system is crucial for addressing the data movement bottleneck. This hierarchy commonly consists of a large off-chip DRAM, a smaller on-chip global buffer (GLB) shared by the PEs, and the aforementioned local memory within each PE.
3) Network-on-Chip (NoC): The movement of data between these memory levels and among the PEs is managed by an on-chip interconnect, or NoC. The NoC is an indispensable component responsible for supporting the communication patterns required by a given dataflow, including unicast, broadcast, and multicast [5].

E. Dataflow

A *dataflow* is the high-level strategy for orchestrating computation and data movement within an accelerator [8]. In the context of the loop nest abstraction, the dataflow is determined by the ordering of the loops. This ordering dictates which operand is held *stationary* in memory for the longest duration, thereby maximizing its temporal reuse and minimizing costly accesses to higher levels of the memory hierarchy [5].

Common dataflows are named after the operand they keep stationary. For instance, a Weight Stationary (WS) dataflow places weight-related loops at the outermost positions, ensuring a single weight tile is reused across many different input activations. Conversely, an Input Stationary (IS) dataflow prioritizes the reuse of input activation tiles, while an Output Stationary (OS) dataflow focuses on accumulating partial sums for a single output tile. This strategic decision can be applied independently at each level of the memory hierarchy [5].

F. Mapping

While a dataflow defines the high-level strategy, a *mapping* provides a more concrete execution plan for a given workload on a specific hardware. A mapping is defined by a set of decisions that instantiate a dataflow, which are directly related to the loop nest abstraction [5], [6]:

1) Dataflow Selection: The choice of loop ordering at each level of the memory hierarchy.
2) Tiling: The loop bounds that partition the workload into tiles.
3) Spatial Unrolling: The assignment of specific loop iterations to be executed spatially across the PE array.

The set of all valid mappings for a given workload and architecture constitutes a vast and complex *mapspace*. The impact of a mapping is significant, as different choices can yield dramatically different latency and energy results. For instance, analyses have shown that energy efficiency can vary by up to 19× for a similar latency [6], making the search for a (near) optimal configuration critical. Navigating this vast mapspace requires automated tools, which need two key components: models to accurately estimate the latency, energy, and area cost of a given mapping; and mappers, which are search algorithms that leverage these models to efficiently explore the mapspace and identify optimal configurations. These modeling and mapping frameworks, which form the core of this report, are therefore essential for the design and fair comparison of accelerator hardware.

III. Foundational Modeling & Mapping Tools

A. Accelergy: A Modular Energy Estimation Framework

As established, navigating the accelerator design space requires fast and accurate cost models. Although post-layout simulation provides ground-truth energy figures, it is far too slow and occurs too late in the design process to be useful for early-stage exploration. In contrast, existing architecture-level estimators such as McPAT [9] were developed for general-purpose processors and lack the necessary flexibility to model the diverse and rapidly evolving landscape of domain-specific accelerators, as they rely on a fixed set of predefined components [10].

Accelergy is an energy estimation framework that addresses this gap. Its core innovation is a modular and extensible approach that separates the architectural description from the underlying energy characterizations. This is achieved through three key concepts.

First, Accelergy employs a hierarchical, object-oriented approach to define architectures. A design is described using a combination of *primitive components* and *compound components*. Primitives are fundamental, indivisible hardware blocks (e.g., a MAC unit, an SRAM bank, a wire) that form

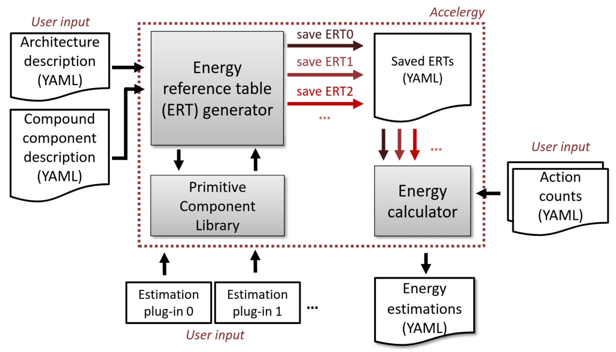


Fig. 4: High-level block diagram of Accelergy framework [10].

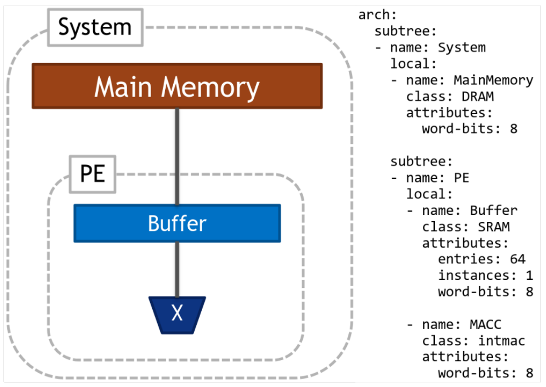
a reusable library. Compound components are higher-level, user-defined modules constructed from primitives and/or other compound components (e.g., a PE, which contains registers and a MAC). This ability for users to define their own compound components is what grants Accelergy the flexibility to model novel and complex accelerator designs.

Second, the framework is technology-agnostic through the use of estimation plug-ins. Accelergy does not contain built-in energy models. Instead, it sources the energy cost of primitive components by querying external tools or data sources through a standardized interface. These plug-ins can be technology-specific lookup tables, analytical models such as CACTI [11] for memories, or data from other simulators. This allows a single architectural description to be evaluated across different process technologies or with updated energy models without modification.

Third, for high accuracy, Accelergy moves beyond simple access counting by defining a rich space of actions. It recognizes that the energy-per-access is highly dependent on how a component is used. It models energy with fine granularity considering factors such as the action property (e.g., read vs. write), data property (e.g., accessing random vs. repeated data), the effects of clock gating, and design-specific optimizations like zero-gating on MAC units [10].

The workflow, illustrated in Figure 4, is a two-phase process. In the first phase, the *Energy Reference Table (ERT)* Generator parses the hierarchical architecture and component descriptions and invokes the necessary plug-ins to generate an ERT. The ERT is a static database that contains the energy-per-action for every component in the design. In the second phase, the Energy Calculator takes this pre-generated ERT and combines it with workload-specific *action counts* (typically produced by a performance simulator) to compute the total energy. This separation is critical for fast exploration: the potentially slow ERT generation is done only once per architecture, while the lightweight energy calculation can be run repeatedly for different workloads or mappings.

In summary, Accelergy provides a flexible energy cost model, establishing a foundation for systematic accelerator comparison. Its role is to evaluate the energy cost for a given mapping. It does not search for an optimal mapping, nor does



```
arch:
  subtree:
  - name: System
    local:
    - name: MainMemory
      class: DRAM
      attributes:
        word-bits: 8

    subtree:
    - name: PE
      local:
      - name: Buffer
        class: SRAM
        attributes:
          entries: 64
          instances: 1
          word-bits: 8

      - name: MACC
        class: intmac
        attributes:
          word-bits: 8
```

Fig. 5: Example architecture specification in Timeloop [12].

it model performance metrics such as latency, nor generate the action counts it requires as input. These capabilities are essential for a holistic accelerator evaluation and motivate the need for a complementary framework.

B. Timeloop: Systematic Mapspace Exploration

While an energy cost model like Accelergy provides a foundation, a holistic evaluation workflow must also address three key challenges that fall outside its scope: first, the search for an optimal mapping within the vast mapspace; second, the modeling of performance metrics such as latency; and finally, the generation of the workload-specific action counts that the energy model requires as input. Timeloop is a framework engineered to solve these challenges [6], automating the search for high-quality mappings through a systematic methodology built on three core abstractions:

1) Workload: Any DNN operation is represented as a loop nest. This generalization allows Timeloop to model various computations, such as convolutions and matrix multiplications, in a unified manner.
2) Architecture: The hardware is described as a hierarchical tree of components, as shown for a simple architecture in Figure 5. This template defines the memory hierarchy, from the PEs' local register files up to off-chip DRAM.
3) Mapping: As defined in Section II-F, a mapping connects the abstract workload loop nest to the physical hardware architecture through a set of tiling, ordering, and spatial unrolling directives.

The mapspace is constructed by enumerating all possibilities within three distinct sub-spaces, each defined with respect to the memory hierarchy. The first is the *IndexFactorization* subspace, which contains all possible ways to factor the bounds of each workload dimension across the different memory levels. The second is the *LoopPermutation* subspace, which includes all valid orderings of the loops scheduled at each level. The third is the *LevelBypass* subspace, which enables tensors with minimal temporal reuse to bypass intermediate levels of the memory hierarchy, thus preserving valuable buffer capacity for higher-reuse data. The complete mapspace is the Cartesian

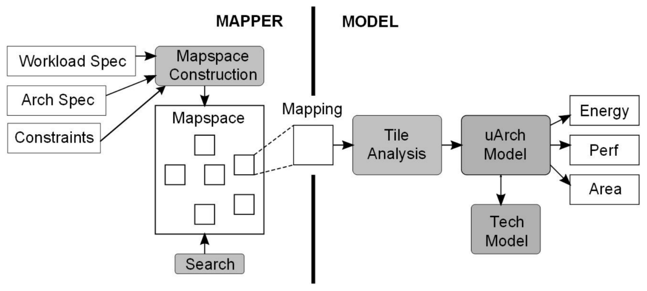
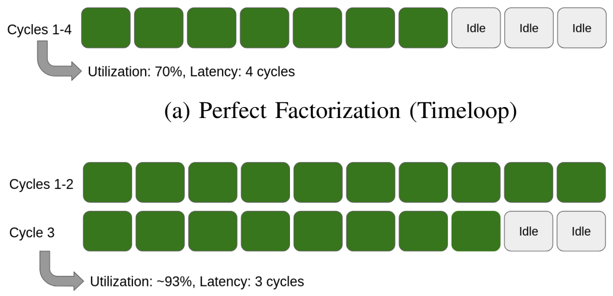


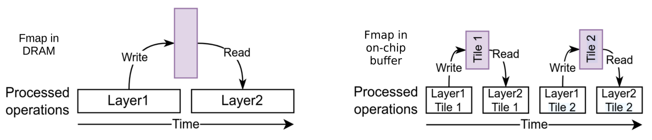
Fig. 6: Timeloop workflow [6].


Cycles 1-4
Idle
Idle
Idle
Utilization: 70%, Latency: 4 cycles
Cycles 1-2
Cycle 3
Idle
Idle
Utilization: ~93%, Latency: 3 cycles

(a) Perfect Factorization (Timeloop)

(b) Imperfect Factorization (Ruby)

Fig. 7: Comparison of perfect vs. imperfect factorization for mapping a workload dimension of size 28 onto a 10-PE array.



(a) Layer-by-layer processing

(b) Fused-layer processing

Fig. 8: Comparison of multi-layer processing strategies. When intermediate feature maps are too large to fit in on-chip buffers, (a) layer-by-layer must write them to off-chip DRAM. (b) fused-layer avoids this by processing in tiles that fit in on-chip buffer [4].

product of these subspaces, which can lead to a combinatorial explosion. Users can prune this space by providing constraints that enforce specific dataflows or model hardware limitations.

The Timeloop workflow, depicted in Figure 6, is centered on a mapper that navigates this mapspace, guided by a fast analytical cost model. The mapper employs search heuristics to sample candidate mappings. The key to Timeloop's efficiency lies in its cost model, which avoids slow, cycle-level simulation. By leveraging the deterministic mathematical structure of the loop nest, it performs a hierarchical tile analysis to calculate the required data movement between adjacent levels of the memory hierarchy. The output of this process is a complete list of action counts: the number of MAC operations, the number of reads and writes for each memory component, and the number of unicast or multicast transfers on the NoC.

From these action counts, Timeloop derives the final performance and energy metrics. Latency is determined by identifying the component (compute, memory, or network) that imposes the primary performance bottleneck. The action counts are also passed to an energy model like Accelergy, which provides the corresponding energy-per-action costs to compute a total energy estimate. In essence, Timeloop makes accelerator design more systematic by decoupling the hardware architecture from the mapping strategy and automating the search for an optimal configuration. While this foundational framework is powerful, its abstractions — namely its assumptions of dense computations, perfect tiling factorization, and a single-layer scope — present clear opportunities for further optimization, which will be explored in the following sections.

IV. Expanding the Mapspace

A. Ruby: Tiling with Imperfect Factorization

Timeloop schedules computations using perfect factorization. This means that the total number of loop iterations for a given dimension must be divisible without remainder by the tiling factors chosen for each level of the memory hierarchy. This constraint can lead to significant hardware underutilization when workload dimensions do not align with the number of PEs. Ruby addresses this limitation by expanding the mapspace to include *imperfect factorization* [13]. This technique partitions a parallelized loop into a main body of full-sized iterations and a final, smaller "cleanup" iteration to process the remainder. As illustrated in Figure 7, this allows for mappings that better conform to the physical hardware. In the example shown, imperfect factorization reduces latency from 4 to 3 cycles while increasing average PE utilization from 70% to approximately 93%.

B. LoopTree: Increase Data Reuse by Fusing Layers

Timeloop is limited by its single-layer scope, confining its analysis to intra-layer data reuse. It treats each layer of a DNN as an independent task. Therefore, it cannot reason about the producer-consumer relationship between them. Consequently, when an intermediate *feature map* (fmap) is too large to be fully retained in a capacity-constrained on-chip buffer, the default execution model results in a costly round-trip: the fmap is written to off-chip DRAM before being read back by the subsequent layer, preventing efficient inter-layer data reuse.

The core principle for addressing this limitation is to establish producer-consumer proximity for intermediate data. This forms the basis of a *fused-layer dataflow*, which, as illustrated in Figure 8, partitions the workload into tiles. Doing so allows an intermediate fmap tile to be passed directly from a producer layer to a consumer layer via a fast on-chip buffer, thereby eliding the costly round-trip to off-chip DRAM [4].

LoopTree extends Timeloop by enabling the holistic optimization of a user-defined *fusion set*, a group of consecutive layers to be fused together. This approach alters the mapping search to operate on the entire set rather than on individual layers. The complexity of this multi-layer search is made tractable by a key abstraction: the mapper explores different inter-layer tiling choices only for the final layer in the fusion set.

For each candidate tiling, LoopTree analytically infers the corresponding inter-layer tile requirements for all preceding layers in the set, by propagating data dependencies backward. This inference defines the boundaries for a set of smaller, independent intra-layer mapping problems, one for each layer. LoopTree supports a full mapping search for this intra-layer execution, enabling a hierarchical co-optimization of inter- and intra-layer computation. The mapspace is further expanded by formalizing the trade-off between retaining an intermediate fmap/tile and recomputing it on-demand.

LoopTree provides the capability to systematically exploit inter-layer data reuse by navigating the trade-offs inherent to layer fusion. However, it has a crucial limitation: the fusion set is a static, user-defined input, leaving the search for an optimal grouping to the hardware architect or future work.

C. Layoutloop: Co-optimizing Dataflow & On-Chip Data Layout

A key limitation of Timeloop is its abstract memory model, which ignores physical data layout and *banking*. This creates a theory-practice gap, where mappings that appear optimal are severely slowed down by *memory bank conflicts* on physical hardware [8]. Layoutloop addresses this by modeling the on-chip buffer as a logical 2D array ($num\_lines \times line\_size$) that is physically partitioned into independently accessible sub-arrays called banks, each with a limited number of access ports. A bank conflict occurs if a computation attempts to simultaneously access more data from a single bank than its ports can service, forcing a memory stall.

Layoutloop expands the original mapspace by introducing the data layout as a new searchable dimension. A specific layout within this expanded space is formally defined by the notation `(Inter-line order)_(Intra-line order)`, such as `CHW_W4H2C2`. Here, the `CHW` prefix dictates the inter-line order for sequencing through tensor dimensions across logical lines. The `W4H2C2` suffix defines the intra-line order for packing data into each line, resulting in a line size of $4 * 2 * 2 = 16$ elements.

Leveraging this specification, Layoutloop performs a per-bank conflict assessment. It calculates a performance slowdown for any bank where the number of simultaneous access requests exceeds the available ports. The overall system latency is dictated by the worst-case slowdown across all banks, enabling a true co-search for a physically realizable and optimal (`dataflow`, `layout`) pair.

V. Modeling Advanced Accelerator Paradigms

A. Sparseloop: Exploration of Sparse Tensor Accelerators

Timeloop is limited by its assumption of dense tensor computations. It models every multiply-accumulate (MAC) operation in a loop nest, even those involving zero-valued operands, which are known as *ineffectual computations*. This prevents it from capturing the primary optimization in sparse accelerators: avoiding the costly execution of these operations to save both energy and latency. To address this gap, Sparseloop extends Timeloop by enhancing its input specifications and its core analytical model [14]. This is achieved through a taxonomy that describes the hardware techniques used in sparse accelerators. This structured classification enables Sparseloop to model the impact of these techniques on latency and energy.

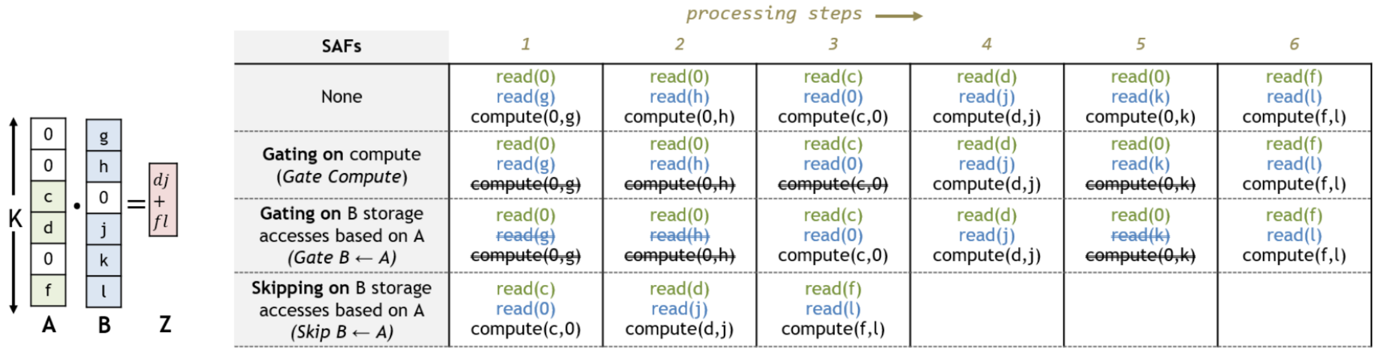
Sparseloop classifies these techniques into three orthogonal categories, termed *Sparsity-Aware Features* (SAFs). The first, *representation format*, governs how sparse tensors are stored, using compressed formats that encode only nonzero values and their metadata to reduce memory footprint and data movement energy. The other two categories, *gating* and *skipping*, address the ineffectual computations directly. The critical distinction between them is illustrated in Figure 9 for a simple dot-product workload. As shown, gating is an energy-saving technique that keeps hardware components idle but does not reduce the overall cycle count. In contrast, skipping is a more powerful optimization that saves both energy and latency by allowing the hardware to completely bypass cycles corresponding to ineffectual operations.

Sparseloop integrates into the Timeloop workflow as an additional modeling stage that post-processes the results of the dense analysis. To enable this, the user must provide two new specifications. First, the workload description is augmented to include statistical density models for each tensor, describing the probability of nonzero values (i.e., the density) and their distribution (e.g., uniform, structured). Second, a new architecture input file defines the SAFs supported at each hardware level. For example, a leader-follower skipping mechanism is denoted by the notation $skip[B] \leftarrow A$, where A acts as the leader: if its value is zero, access to the follower (B) is skipped.

Equipped with this information, Sparseloop takes the dense action counts and latency projections generated by the baseline Timeloop model. It then analytically applies the specified SAFs, using the tensor density models to calculate the probable number of operations that will be gated or skipped. This process adjusts the dense action counts into final sparse action counts and recalculates the system latency, yielding a fast and accurate projection of the sparse accelerator's performance and energy. By providing this systematic way to describe and model sparsity, Sparseloop enables a fair comparison of sparse accelerators.

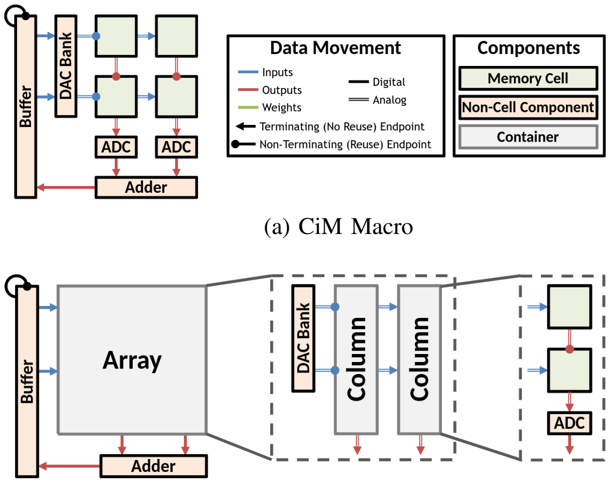
B. CiMLoop: Exploration of Compute-in-Memory Architectures

Compute-in-Memory (CiM) has emerged as a promising accelerator paradigm designed to overcome the data movement bottleneck. By performing computations directly within the memory array itself, CiM architectures avoid energy-intensive data movement and leverage the high density of memory arrays to perform parallel, low-energy computations. These operations are often performed in the analog domain, which introduces a fundamental modeling challenge: unlike digital logic, the energy consumption of analog circuits is highly data-value-dependent.



SAFs	1	2	3	4	5	6
None	read(0) read(g) compute(0,g)	read(0) read(h) compute(0,h)	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(0) read(k) compute(0,k)	read(f) read(l) compute(f,l)
Gating on compute (*Gate Compute*)	read(0) read(g) ~~compute(0,g)~~	read(0) read(h) ~~compute(0,h)~~	read(c) read(0) ~~compute(c,0)~~	read(d) read(j) compute(d,j)	read(0) read(k) ~~compute(0,k)~~	read(f) read(l) compute(f,l)
Gating on B storage accesses based on A (*Gate B ← A*)	read(0) ~~read(g)~~ ~~compute(0,g)~~	read(0) ~~read(h)~~ ~~compute(0,h)~~	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(0) ~~read(k)~~ ~~compute(0,k)~~	read(f) read(l) compute(f,l)
Skipping on B storage accesses based on A (*Skip B ← A*)	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(f) read(l) compute(f,l)			

Fig. 9: Processing a sparse dot-product with different Sparsity-Aware-Features (SAFs). 1st row: baseline processing without SAFs; 2nd row: Gating applied to compute; 3rd row: Gating applied to B reads based on A's values; 4th row: Skipping applied to B reads based on A's values [14].



(a) CiM Macro

(b) Hierarchy of Containers

Fig. 10: CiMLoop's Container-Hierarchy partitions systems into a hierarchy of user-defined containers [15].

The foundational Timeloop and Accelergy frameworks are ill-equipped to model CiM systems due to two key limitations. First, their strict separation of memory and computation, reflected in Timeloop's topological - and Accelergy's compositional hardware representation, cannot represent a system where computation occurs inside a memory array. Second, and more critically, Accelergy's fixed-energy-per-action assumption is incompatible with the data-value-dependent nature of analog CiM components like Digital-to-Analog (DACs) and Analog-to-Digital Converters (ADCs).

CiMLoop is a framework that extends the Timeloop-Accelergy ecosystem to solve these specific problems [15]. It achieves this through two core innovations: (1) a unified and flexible hardware description model, and (2) an enhanced energy modeling pipeline that supports data-value dependency.

To address the structural limitation, CiMLoop replaces the separate hardware descriptions with a single, flexible *Container-Hierarchy*. This user-defined, nested structure allows compute primitives to be placed inside memory-like containers, mirroring the physical hardware. For instance, a user can model a CiM Array container that physically contains DAC, ADC, and Memory Cell components. As illustrated in Figure 10, this approach enables a systematic decomposition of a complex macro, capturing both its physical composition and topological relationships in one unified specification.

To capture the data-value-dependent energy of analog circuits, CiMLoop makes a fundamental extension to Accelergy's core modeling engine. Its plug-in interface is enhanced so that component models can accept statistical distributions of operand values as an input, in addition to the standard action type. This allows for the calculation of an accurate average energy without the prohibitive slowdown of simulating every individual data value, thereby achieving both speed and accuracy for analog circuit modeling.

By unifying the hardware description and introducing a data-value-dependent energy model, CiMLoop provides a framework for systematic exploration and comparison of CiM architectures.

TABLE I: Overview of Timeloop Extensions

Framework	Core Problem	Key Innovation
Ruby [13]	PE Underutilization	Imperfect Tiling
LoopTree [4]	Inter-Layer Data Movement	Fused-Layer Co-optim.
Layoutloop [8]	Memory Bank Conflicts	Data Layout Co-optim.
Sparseloop [14]	Ineffectual Computations	Sparsity-Aware Modeling
CiMLoop [15]	Analog CiM Modeling	Data-Aware Energy Models

VI. Future Directions

The primary challenge in automated accelerator design is a fundamental disconnect: key stages of the design process, including Neural Architecture Search (NAS), mapping frameworks like Timeloop, and compiler implementation, each operate on a different, often conflicting, set of assumptions about performance and cost. This fragmentation leads to suboptimal design choices, as the assumptions made at one stage - whether in hardware design, mapping exploration, or model creation - are often inconsistent with the realities of the others.

The ultimate vision, articulated in works like Bringmann et al. [16], is to resolve this fragmentation with a single, holistic HW/SW co-design process. This is achieved through an iterative search loop where high-level decisions are guided by low-level realities. In this loop, a high-level tool proposes a candidate, which is then rapidly analyzed by lowering it through a unified compiler infrastructure like MLIR [17]. A realistic cost estimate is extracted from this lowered state and used as a feedback signal to guide the next search iteration.

While this unified loop is the long-term goal, the immediate research path involves building its key components. A primary sub-goal is expanding the search space to include the hardware architecture itself. Frameworks like DiGamma [18] and SENNA [19] are pioneering this by co-searching over architectural parameters alongside mappings to find the optimal hardware for a given workload. The next step is the deeper integration of these hardware searchers, NAS tools, and mapping explorers into a single, cohesive framework, creating a true co-design engine where cost models are consistent from algorithm to machine code.

VII. Conclusion

The demand for efficient ML execution on embedded devices has driven the development of domain-specific accelerators, but the vastness of the associated design space makes their creation a formidable task. Comparing architectures also presents a significant challenge, as performance is highly dependent on the software mapping strategy.

This report has surveyed the evolution of modeling and mapping frameworks that bring a systematic, scientific rigor to this challenge. We began with the foundational Timeloop and Accelergy ecosystem, which established the core principle of decoupling hardware architecture from mapping exploration to enable fair and robust comparisons. We then examined the subsequent wave of innovation, showing how the field has expanded upon this base. Frameworks like Ruby, LoopTree, and Layoutloop have progressively enlarged the mapspace to capture more realistic and efficient execution strategies by incorporating imperfect tiling, inter-layer fusion, and physical data layout. Concurrently, tools like Sparseloop and CiMLoop have extended modeling capabilities to capture advanced architectural paradigms that defy traditional assumptions.

Together, these frameworks represent a paradigm shift in hardware design. They are transforming the process from one reliant on expert intuition into a systematic, data-driven engineering discipline. By providing the means to rapidly evaluate trade-offs, identify bottlenecks, and quantify the benefits of new architectural features, these tools not only find optimal mappings for existing hardware but also guide the design of future accelerators, paving the way towards the ultimate goal of fully automated hardware-software co-design.

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[2] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016.

[3] Y. Jia, *Learning semantic image representations at a large scale*. University of California, Berkeley, 2014.

[4] M. Gilbert, Y. N. Wu, J. S. Emer, and V. Sze, "Looptree: Exploring the fused-layer dataflow accelerator design space," *IEEE Transactions on Circuits and Systems for Artificial Intelligence*, 2024.

[5] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient processing of deep neural networks*. Springer, 2020.

[6] 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 *IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 304–315, IEEE, 2019.

[7] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pp. 10–14, IEEE, 2014.

[8] J. Tong, A. Itagi, P. Chatarasi, and T. Krishna, "Feather: A reconfigurable accelerator with data reordering support for low-cost on-chip dataflow switching," in *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 198–214, IEEE, 2024.

[9] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, pp. 469–480, 2009.

[10] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2019.

[11] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 694–701, IEEE, 2011.

[12] A. Parashar *et al.*, "Tutorial: Timeloop and accelergy: Automating dnn accelerator modeling and analysis." Tutorial presentation at IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2020. Available at https://accelergy.mit.edu/ispass2020/2020_08_23_timeloop_accelergy_tutorial_part1.pdf.

[13] M. Horeni, P. Taheri, P.-A. Tsai, A. Parashar, J. Emer, and S. Joshi, "Ruby: Improving hardware efficiency for tensor algebra accelerators through imperfect factorization," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 254–266, IEEE, 2022.

[14] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical approach to sparse tensor accelerator modeling," in *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1377–1395, IEEE, 2022.

[15] T. Andrulis, J. S. Emer, and V. Sze, "Cimloop: A flexible, accurate, and fast compute-in-memory modeling tool," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 10–23, IEEE, 2024.

[16] O. Bringmann, W. Ecker, I. Feldner, A. Frischknecht, C. Gerum, T. Hämäläinen, M. A. Hanif, M. J. Klaiber, D. Mueller-Gritschneder, P. P. Bernardo, *et al.*, "Automated hw/sw co-design for edge ai: State, challenges and steps ahead," in *Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis*, pp. 11–20, 2021.

[17] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, IEEE, 2021.

[18] S.-C. Kao, M. Pellauer, A. Parashar, and T. Krishna, "Digamma: Domain-aware genetic algorithm for hw-mapping co-optimization for dnn accelerators," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 232–237, IEEE, 2022.

[19] J. Kwon, H. Min, and B. Egger, "Senna: Unified hardware/software space exploration for parametrizable neural network accelerators," *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 2, pp. 1–26, 2025.