DOSA: Differentiable Model-Based One-Loop Search for DNN Accelerators

Charles Hong
University of California, Berkeley
Berkeley, CA, USA
charleshong@berkeley.edu

Qijing Huang
NVIDIA
Santa Clara, CA, USA
jennyhuang@nvidia.com

Grace Dinh
University of California, Berkeley
Berkeley, CA, USA
dinh@berkeley.edu

Mahesh Subedar
Intel Labs
Hillsboro, OR, USA
mahesh.subedar@intel.com

Yakun Sophia Shao
University of California, Berkeley
Berkeley, CA, USA
ysshao@berkeley.edu

ABSTRACT

In the hardware design space exploration process, it is critical to optimize both hardware parameters and algorithm-to-hardware mappings. Previous work has largely approached this simultaneous optimization problem by separately exploring the hardware design space and the mapspace—both individually large and highly nonconvex spaces—independently. The resulting combinatorial explosion has created significant difficulties for optimizers.

In this paper, we introduce DOSA, which consists of differentiable performance models and a gradient descent-based optimization technique to simultaneously explore both spaces and identify high-performing design points. Experimental results demonstrate that DOSA outperforms random search and Bayesian optimization by 2.80× and 12.59×, respectively, in improving DNN model energy-delay product, given a similar number of samples. We also demonstrate the modularity and flexibility of DOSA by augmenting our analytical model with a learned model, allowing us to optimize buffer sizes and mappings of a real DNN accelerator and attain a 1.82× improvement in energy-delay product.

CCS CONCEPTS

• **Hardware** → **Hardware-software codesign**; **Application specific processors**; • **Computing methodologies** → **Machine learning**; **Modeling and simulation**; **Search methodologies**.

KEYWORDS

Design space exploration, Machine learning accelerators

ACM Reference Format:
Charles Hong, Qijing Huang, Grace Dinh, Mahesh Subedar, and Yakun Sophia Shao. 2023. DOSA: Differentiable Model-Based One-Loop Search for DNN Accelerators. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3623797

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3623797

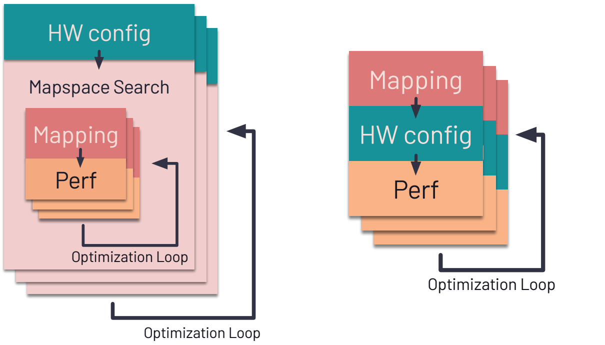


Figure 1: Hardware-first, two-loop (left) and mapping-first, one-loop (right) DSE approaches.

1 INTRODUCTION

Deep neural network (DNN) accelerators [2, 12, 23, 32] have become a critical driving force for the recent breakthroughs [7, 8, 20, 33, 42] in artificial intelligence. To develop efficient DNN accelerators in a fast and cost-effective manner, automated design space exploration (DSE) has emerged as a powerful technique.

The hardware DSE flow [22, 34, 43] involves optimizing over two search spaces: the *hardware design space*, which describes hardware design parameters such as interconnect topology and buffer and systolic array sizes, and the *mapspace*, which describes how applications are executed on the target hardware and encompasses decisions such as tiling, dataflow, and spatio-temporal mapping.

For both the hardware design space and the mapspace, the goal is to optimize a performance metric, such as energy-delay product (EDP), subject to certain constraints. These include design budgets, such as bounds on the area or power consumption, as well as constraints ensuring that the selected mapping can be executed on the selected hardware configuration (e.g. that the hardware buffers are sufficiently large to contain the tiles). As these constraints encompass both the mapspace and the hardware design space, the two spaces must be simultaneously optimized over; techniques solely tackling hardware search [21, 53] or mapping optimizations [1, 3, 4, 19, 31, 35, 37] are insufficient to achieve the optimal hardware design in DSE.

Both the hardware design and mapping spaces are vast, high-dimensional, and comprised of both categorical and discrete variables. Furthermore, evaluating the performance of a hardware configuration and a mapping can be computationally expensive. The size of the combined optimization space and the cost of evaluating points in it pose formidable challenges to DSE algorithms.

Much prior work [10, 26, 38, 43, 50, 54] has approached this problem using *hardware-first search*. These methods directly search over the space of possible hardware configurations. The performance of each hardware configuration is calculated by first constraining the mapspace to mappings that are compatible with the hardware configuration, then optimizing over the constrained (highly discontinuous) mapspace. In most cases, the mapspace optimization is done iteratively, rendering this process a *two-loop approach* iterating over both the hardware space and mapspace. As a result, these approaches must contend with a combinatorial explosion of possible configurations.

Alternatively, *mapping-first* approaches, as proposed in [15, 52] and illustrated in Figure 1, optimize primarily over the mapspace. For each mapping, optimizing over the hardware design space is a straightforward process consisting of finding the minimal hardware configuration capable of supporting the mapping. As a result, the loop for hardware search is eliminated, allowing the entire DSE process to be encapsulated in a single loop. Furthermore, the lack of hardware resource constraints also significantly simplifies the mapspace search problem.

Despite these advantages, mapping-first approaches must still contend with the size of the mapspace and the nonconvexity of the performance over this space. Prior works have either directly applied black-box optimization methods [15, 40], which rely on a large number of (often expensive to collect) samples, or pruned the search space using architecture-specific heuristics constructed by hand based on observations of a limited subset of the DSE search space [52].

Reducing the sample complexity of DSE while still allowing for a systematic exploration of the entire space requires leveraging domain knowledge—for instance, a generalizable performance model like Timeloop [34, 47] (a popular analytical model for DNN accelerators), that is not reliant on an expensive training process. This paper follows this approach, using performance models as an optimization target for mapping-first search. Specifically:

- We build closed-form *differentiable* and interpretable performance models for latency and energy on DNN accelerators. Our models are as precise as the state-of-the-art program-based analytical models, while also being amenable to white-box optimization techniques such as gradient descent.
- We then introduce DOSA[1], a mapping-first one-loop DSE flow that uses gradient descent to find the most efficient hardware parameters and mappings to target multi-layer DNNs; to the best of our knowledge, this is the first work to use a mapping-first strategy to simultaneously perform DSE for multiple layers of a neural network. DOSA converges at least 40% faster than state-of-the-art DSE approaches.

	Name	**Mapspace Search**	**Hardware Search**
Two-loop Searchers	Spotlight [38]	BB-BO	BB-BO
	VAESA [10]	ILP [11]	VAE+BB-BO/GD
	FAST [54]	BB-LCS [17]+ILP	BB-LCS
	HASCO [50]	RL	BB-BO
	NAAS [26]	BB-ES	BB-ES
	MAGNet [43]	Heuristics	BB-BO
One-loop Searchers	DiGamma [15]	BB-GA	
	Interstellar [52]	Heuristics	
	Our work: DOSA	**GD**	

Table 1: State-of-the-art Accelerator DSE Methods.

- We take a step beyond DSE for architectural model by introducing a DNN model to predict the variation between analytical model and real hardware accelerator performance, and using it to augment our differentiable model for real hardware DSE.
- We benchmark our results on the Gemmini accelerator, showing a 1.82× EDP improvement over hand-designed configurations.

2 BACKGROUND

Hardware design space exploration (DSE) is a time-consuming and costly process that involves the exploration of various hardware design parameters and software mappings to optimize the target application performance. This process typically includes two key optimizations: the mapping search, which aims to find high-performance mappings that effectively utilize hardware resources, and the hardware search, which aims to achieve multi-objective design goals, such as minimizing the energy-delay product (EDP) or the area-delay product. To address the mapping complexity for DNNs, many DNN compilers [1, 3, 4, 19, 31, 35, 37] and accelerator-aware mapping techniques [9, 11, 13, 24, 34, 52] have been developed. In addition, there has been extensive research in the area of hardware parameter search [21, 53].

2.1 Co-exploration Frameworks

In recent years, there has been a growing body of research focused on tackling the compounding search space of mapping and hardware designs with the goal of achieving higher hardware efficiency and lower development costs.

2.1.1 Two-Loop Searchers. As shown in Table 1, most prior work [10, 26, 38, 43, 50, 52, 54] treats the mapping and hardware co-search as a two-loop process and applied a combination of various optimization techniques to address each search space independently. The two-loop process starts by sampling a hardware design point from the hardware search space and then searching for high-performance mappings for that particular hardware design point in the inner loop. The best mapping obtained is used for generating the hardware performance feedback for the outer loop hardware optimization.

Optimization techniques used in the two-loop process can be broadly categorized into three types: heuristics, black-box optimization (BB), and white-box optimization. Heuristics involve using

[1] Code open-sourced at https://github.com/ucb-bar/dosa.
Artifacts available: see Appendix A.

domain-specific knowledge and experience to guide the search process and reduce the size of design space. In contrast, BB relies on sampling and machine learning techniques to leverage the characteristics of the problem derived from sampled data in order to find the optimal solution. Popular BB algorithms include genetic algorithms (BB-GA), reinforcement learning (RL), Bayesian optimization (BB-BO), Linear Combination Swarm (BB-LCS), and evolutionary strategy (BB-ES).

In white-box optimization, the relationship between the optimization variables and the objectives is known and captured in mathematical models. Numerical optimization techniques like linear programming (LP) and mixed-integer programming (MIP) can be used if the relationship can be expressed in specific frameworks. Gradient descent (GD) techniques can be applied if the relationship can be expressed in a differentiable expression. Compared to black-box optimization, white-box optimization is generally more efficient as it can exploit the known objective model to guide the optimization process, resulting in faster convergence. However, it requires the objective model to be known and accurately specified.

While independently applying optimization techniques to the mapspace and hardware space can be effective, the two-loop searchers can be susceptible to combinatorial explosion, as the vast search space multiplies the number of potential options for mappings and hardware parameters together.

2.1.2 Single-Loop Searchers. To reduce the size of the compounding search space, one-loop searchers, such as DiGamma [15] and Interstellar [52] have been proposed. Single-loop search tackles the co-search problem from a mapping-first approach that infers the minimal hardware requirement from hardware-agnostic high-performance mappings found in single-loop mapping search. In such approaches, the hardware DSE space is similar in size to the mapping space. However, DiGamma employs BB-GA which treats the mapping performance as a black-box and needs to evaluate many unique hardware and mapping configurations iteratively to achieve a good mapping and hardware design. Interstellar, on the other hand, only explores a limited space of pre-selected mappings and as a result only a limited space of hardware design is evaluated.

Unlike previous one-loop approaches that rely on black-box optimizations or heuristics, DOSA takes a novel approach by formulating the analytical performance and energy model in [34] as a differentiable white-box model. DOSA uses gradient descent to optimize the mapping variables in the direction of steepest descent of the EDP objective function on the mathematical model. This allows DOSA to explore a comprehensive set of mappings and efficiently generate high-quality hardware and mapping configurations without the need for sampling from simulators extensively.

2.2 Performance Modeling

Performance models are crucial to the DSE process, as they offer quick feedback on performance and energy consumption for different hardware and mappings. They provide valuable insights into how hardware designs perform in real-world scenarios without requiring real hardware prototypes or implementations. They can also reduce the high sampling cost of time-consuming and computationally expensive simulation and emulation for existing hardware designs. In this paper, we illustrate how a well-designed performance model can be used to accelerate the DSE process.

Depending on how they are developed, performance models can be categorized as either analytical models or data-driven models. For DNN accelerators, architects have developed domain-specific analytical models in the form of mathematical equations [11] or iterative programs [22, 27, 28, 34, 48] to quickly assess tradeoffs for various hardware designs. These models leverage workload characteristics (e.g., known iteration space bounds and statically analyzable data access patterns), and hardware characteristics (e.g. roofline model [46]) to perform the estimation. Data-driven models [4, 9, 18], on the other hand, use statistical techniques to fit a machine learning (ML) model to performance data collected over time.

Different models offer different levels of fidelity and compatibility with DSE optimization algorithms. Iterative programs are often used with BB algorithms as the relationship between inputs and predictions is not directly known to the optimization algorithms, which rely on sampling to recapture this relationship. Analytical models expressed in mathematical equations can be used directly as objectives in optimization, but the existing formulations in linear or quadratic programs [11] tend to be limited in expressiveness for complex hardware systems and can result in low accuracy. ML models can be integrated with various optimization techniques easily, but they typically need a large amount of training data to provide accurate prediction and generalize to new workloads and architectures.

This work aims to improve upon existing performance models by introducing a differentiable performance model that is highly accurate, generalizable, and amenable to various efficient white-box optimization algorithms, such as GD. Our approach involves decoding the mathematical relationships from Timeloop [34], an accurate iterative program-based analytical model for DNN accelerators, and converting part of the iterative program into differentiable mathematical models. We show that, by making the objectives differentiable with respect to the design parameters, our approach achieves high sample efficiency in DSE.

While existing analytical models for DNN accelerators are trusted by architects, they may not capture all the interactions between the accelerator and the rest of the hardware systems in real-world deployment. To address this, this work also introduces an ML model that predicts the differences between the analytical model and the actual hardware, thereby improving the effectiveness of DSE in a real-world setting.

3 DOSA OVERVIEW

This paper presents DOSA, a one-loop differentiate-model-based DSE framework to optimize the mappings and hardware simultaneously for target DNN models. DOSA captures key relations between DNN mapping factors and performance objectives in a differentiable analytical model. In addition, DOSA introduces a data-driven DNN model to capture the performance variations between analytical model and real hardware. By applying white-box optimization to the model and calculating the hardware parameters using minimal parameterization, DOSA achieves high-performance

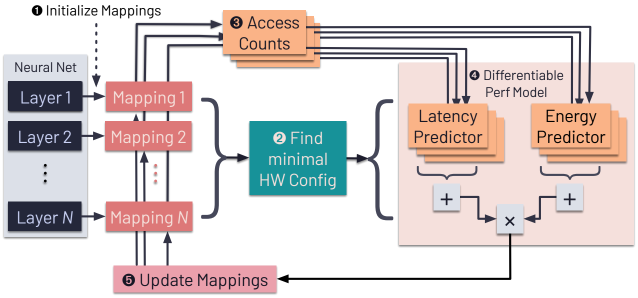


Figure 2: An architecture diagram of DOSA.

accelerator design and mapping while significantly reducing the time and costs associated with DNN accelerator DSE.

3.1 Problem Setup

3.1.1 Target Workloads. DOSA targets accelerator DSE for complete DNN models, which comprise both matrix multiplication and convolution layers. To express these layers, we use seven dimensions: R (weight height), S (weight width), P (output activation height), Q (output activation width), C (input channels), K (output channels), and N (batch size). These dimensions describe the size of the weight (W), input (I), and output (O) tensors. We assume the activation functions are fused with the matrix multiplication and convolution layers.

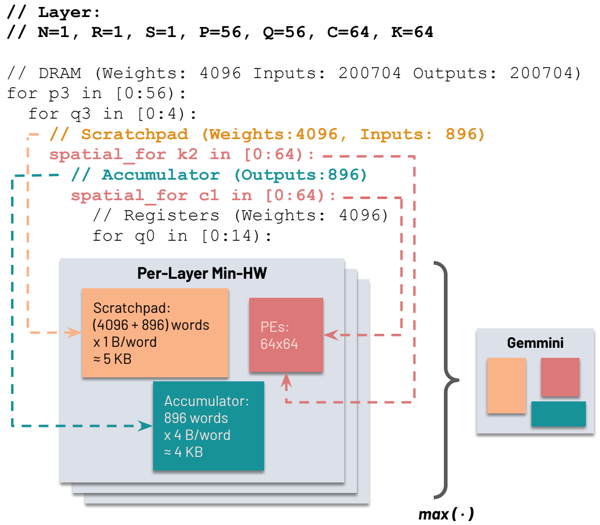
3.1.2 Variables and Objectives. In our mapping-first search, we focus on the following three layerwise mapping decisions:

(1) Spatial loop tiling, which defines which loops are mapped to parallel spatial resources (such as processing elements in a systolic array), and the iteration bounds of these loops,
(2) Temporal loop tiling, which specifies the loop iteration bounds grouped together to form a block at each memory level.
(3) Loop ordering, which defines the order in which dimensions are accessed at a given memory level.

We utilize the spatial and temporal tiling factors, denoted as $\vec{f}$, as input optimization variables in our approach. Specifically, for dimension d at memory level i, $f_{S,i,d}$ and $f_{T,i,d}$ represent the spatial and temporal tiling factors, respectively. Using $\vec{f}$, we construct DOSA's objective function, which serves as the analytical performance model predicting energy-delay product (EDP) of the DNN, as detailed in Section 4. To optimize performance, gradient descent is employed to differentiate the objective with respect to the tiling variables $\vec{f}$. The optimization details are elaborated on in Section 5. Note that there are constraints imposed on the variables to ensure that for each dimension, the product of the spatial and temporal tiling factors at all memory levels is equal to the total problem size.

3.2 Toolflow

Figure 2 provides an overview of how DOSA simultaneously optimizes mappings and hardware for a given workload consisting of a set of layers. The following are the detailed steps involved in this process:



```
// Layer:
// N=1, R=1, S=1, P=56, Q=56, C=64, K=64

// DRAM (Weights: 4096 Inputs: 200704 Outputs: 200704)
for p3 in [0:56):
  for q3 in [0:4):
    // Scratchpad (Weights:4096, Inputs: 896)
    spatial_for k2 in [0:64):
      // Accumulator (Outputs:896)
      spatial_for c1 in [0:64):
        // Registers (Weights: 4096)
        for q0 in [0:14):
```

Figure 3: Mapping to hardware parameters conversion in DOSA. The final hardware configuration is selected by taking the max across mappings for each hardware parameter.

(1) Generate performant mappings using CoSA [11] for a set of target DNN layers, targeting a randomly selected valid hardware design.
(2) Compute the hardware resource requirements of the layer-wise mappings and convert them to a minimal hardware parameterization.
(3) Given the mappings represented in $\vec{f}$, use the differentiable model in DOSA to calculate the number of arithmetic operations and the number of accesses made by each mapping to each memory level in the accelerator.
(4) Combine arithmetic operation and memory access counts with previously calculated hardware parameters to generate roofline-based latency predictions and event-based energy predictions for each layer's mapping. Then, each mapping's latency and energy prediction is combined to produce a single EDP value.
(5) Use gradient descent to update all mappings in parallel.
(6) Repeat from Step 2.

4 DOSA DIFFERENTIABLE MODEL

The differentiable model of DOSA is motivated by the following observations:

- As discussed in Section 2.1, inferring hardware parameters from mappings flattens the hardware-mapping co-search space, and allows for mapping-first, one-loop search.
- An effective mapping-first searcher should co-optimize mapping variables of all layers in the target DNN. This high-dimensional problem requires the application of an efficient optimization method such as gradient descent.
- A differentiable performance model can facilitate DSE with gradient descent, and constructing the model analytically

Arch. Component	Memory Level	Bandwidth (words/cycle)	Energy Per Access (EPA, in uJ) [29]
PE			0.561
Registers	0	$2C_{PE}$	0.487
Accumulator	1	$2\sqrt{C_{PE}}$	$1.94 + 0.1005 \times \frac{C_1}{\sqrt{C_{PE}}}$
Scratchpad	2	$2\sqrt{C_{PE}}$	$0.49 + 0.025 \times C_2$
DRAM	3	8	100

Table 2: Details of the accelerator under study. C_{PE} is the total number of PEs and C_i is the capacity of memory level i.

is preferable as it ensures the accuracy, interpretability, and generalizability of the model.

Given the absence of a differentiable, analytical model for DNN accelerators in current literature, we present our approach for constructing such a model that achieves accuracy on par with Timeloop in our problem space. To account for performance variations in real hardware that are difficult to capture and express in analytical models, we in addition trained a differentiable DNN model to further improve the accuracy of the performance model.

4.1 Computing Hardware Resource Requirements

We target the open-source DNN accelerator Gemmini [6], whose most notable architectural components are 1) a systolic array of processing elements (PEs), 2) accumulator SRAM, 3) scratchpad SRAM, and 4) DRAM. Specifically, we target the weight-stationary (WS) configuration of Gemmini. The buffer levels are enumerated 1, 2, ..., where level 1 represents the buffer memory level. Memory level 0 represents the per-PE registers in the systolic array. The architectural components of Gemmini are further detailed in Table 2. As depicted in Figure 3, the capacity requirements at each level are first computed. Then, we take a parameter-wise max to generate a design that will support all current mappings.

4.1.1 Notation. In our notation, we use i to index memory levels and d to index problem dimensions for the spatial and temporal tiling factors $f_{S,i,d}$ and $f_{T,i,d}$, as listed in Table 3. The spatial or temporal factor is indexed using k in the subsequent section. Additionally, we use t to index each data tensor.

i	memory level index
d	problem dimension index
k	spatial / temporal index
t	data tensor index

Table 3: Notation.

We define the following sets to express the consideration of problem dimensions for calculating the size of data tensors in DNN computations:

$$\begin{aligned} D &= \{R, S, P, Q, C, K, N\} \\ D_W &= \{R, S, C, K\} \\ D_I &= \{R, S, P, Q, C, N\} \\ D_0 &= \{P, Q, K, N\} \end{aligned}$$

The set D contains all the problem dimensions, while D_W, D_I, and D_O are subsets of D that contain the problem dimensions used to calculate the data tensor size and the minimal hardware requirements for weights, inputs, and outputs, respectively.

$$M = \{0, 1, 2, 3\}$$

M is a set of indices that represents the memory levels available for storing intermediate tensors during the computation. To keep track of which tensors are stored at each memory level, we define a matrix B as shown in Table 4. The entries of B indicate whether a tensor with a certain problem dimension is stored at a certain memory level.

		Tensor		
		W	I	O
Registers	0	✓		
Accumulator	1			✓
Scratchpad	2	✓	✓	
DRAM	3	✓	✓	✓

Table 4: Constant binary matrix B, which encodes the data tensor(s) stored at each level of the memory hierarchy, for the accelerator under study.

4.1.2 PE Capacity Requirements. Gemmini supports only square arrays of processing elements. In its WS (weight stationary) configuration, it can parallelize the input channel (C dimension) and output channel (K dimension), each along one side of the array. Hence, we need to configure a square PE array that is large enough to accommodate the square of the larger of these two spatial factors. The total number of processing elements in the systolic array is denoted by C_{PE}.

$$C_{PE} = max(f_{S,1,C}, f_{S,2,K})^2 \tag{1}$$

4.1.3 Buffer Capacity Requirements. Buffer capacities required at a given level i for each tensor are computed by multiplying the related factors $f_{k,j,d}$ together.

$$C_{i,W} = \prod_{(k,j,d)\in\{S,T\}\times\{i-1,i-2,\ldots,0\}\times D_W} f_{k,j,d} \tag{2}$$

$$\text{Inner}(i,d) = \prod_{(k,j)\in\{S,T\}\times\{i-1,i-2,\ldots,0\}} f_{k,j,d}$$

$$\begin{aligned} C_{i,I} = &\left(\prod_{(k,j,d)\in\{S,T\}\times\{i-1,i-2,\ldots,0\}\times\{C,N\}} f_{k,j,d} \right) \\ &\times (Pstride \times (\text{Inner}(i,P) - 1) + \text{Inner}(i,R)) \\ &\times (Qstride \times (\text{Inner}(i,Q) - 1) + \text{Inner}(i,S)) \end{aligned} \tag{3}$$

$$C_{i,O} = \prod_{(k,j,d)\in\{S,T\}\times M\times D_O} f_{k,j,d} \tag{4}$$

$C_{i,t}$ represents the number of words of tensor t that memory level i must be able to hold. Note that to calculate the size required for inputs $C_{i,I}$, we first need to calculate the input activation dimensions using the stride and the output and weight dimensions factors (P,Q,R,S).

The total buffer capacity requirement at level i is the sum of $C_{i,t}$ for each tensor t that is stored at that level:

$$C_i = \sum_{t\in\{W,I,O\}} B_{i,t} C_{i,t} \tag{5}$$

4.2 Traffic Estimation

To capture the performance of the accelerator, we utilize differentiable non-convex functions to model the data movement at each buffer level. We use the following terminologies to refer to different types of data transfer:

- Writes - backing store memory to current memory
- Updates - faster memory or MAC to current memory
- Reads - current memory to faster memory or MAC

4.2.1 Writes. The number of writes to a given memory level i in Gemmini attributable to the tensor t is given by multiplying the tensor size $C_{i,t}$ at level i with all tiling factors outer to the innermost relevant loop, that is outer to level i.

$$\text{Writes}_t(i) = C_{i,t} \prod_{\substack{(k,j,d)\in S,T\times\{i+1,i+2,\ldots,M\}\\ \times D \text{ outer to } D_t>1}} f_{k,j,d} \tag{6}$$

For instance, to calculate the writes to weights, we can multiply the weight tensor size at memory level i, denoted as $C_{i,W}$, with all the tiling factors that are outer to the innermost loops R, S, C, or K given the loop order of all dimensions. This calculation is performed similarly for the outputs. For inputs, as we did for the capacity calculation, we need to first compute the input factors by considering the stride and padding.

The total tensor traffic at a given level is computed by summing weight, input, and output traffic.

4.2.2 Updates. Once write counts are computed, update traffic can be computed more easily. Only outputs and partial sums incur updates to the current memory level i from the inner memory or MAC. For every MAC operation, it will incur an output or partial sum update to the innermost memory level that stores the outputs. Therefore the number of updates to the innermost memory level is equal to the total number of MACs, which is defined as follows:

$$\text{MACs} = \prod_{(k,j,d)\in\{S,T\}\times M\times D} f_{k,j,d} \tag{7}$$

In the Gemmini architecture, as indicated in Table 4, the innermost level corresponds to the accumulator at memory level 1. In the outer levels, the number of updates is equal to the number of writes to the next inner level that holds outputs, as each time a partial sum is loaded, it undergoes addition and is subsequently stored back as an update. Note that the accumulation can also happen in the spatial network which will not result in an update to the memory. The overall reduction to the updates $F_{S,O}(i)$ can be determined by multiplying the spatial factors that are not related to the outputs:

$$F_{S,O}(i) = \prod_{d\in\{D-D_O\}} f_{S,i,d} \tag{8}$$

Combining all these factors, the total updates at memory level i can be expressed as:

$$\begin{gathered}\text{Updates}_O(i) = \begin{cases} \frac{\text{MACs}}{F_{S,O}(i)} & i = \text{innermost output level} \\ \frac{\text{Writes}_O(i-1)}{F_{S,O}(i)} & i > \text{innermost output level} \end{cases} \\ \text{Updates}(i) = B_{i,O}\text{Updates}_O(i)\end{gathered} \tag{9}$$

where the MACs and Writes(i) are discounted by the factors that are spatially accumulated in the network.

4.2.3 Reads. Similarly, when it comes to read operations, in the case of the innermost input buffer that holds inputs, the total number of reads is equal to the total number of MACs. This is because we need to load an input for each MAC calculation. For the outer levels, all the reads from the current level are transferred to the inner level as writes. In the presence of a broadcast spatial network at a certain level, if there are factors $F_{S,t}(i)$ that are irrelevant to the tensor, the same read operation will be broadcasted to different children, eliminating the need for multiple reads:

$$F_{S,t}(i) = \prod_{d\in\{D-D_t\}} f_{S,i,d} \tag{10}$$

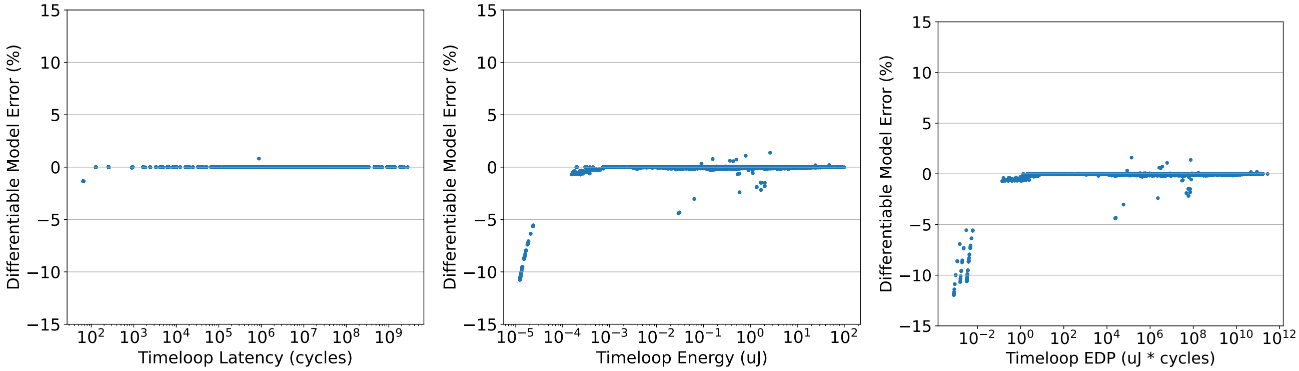
Putting it all together, we have number of reads defined as:

$$\begin{gathered}\text{Reads}_t(i) = \begin{cases} \frac{\text{MACs}}{F_{S,t}(i)} & i = \text{innermost tensor level} \\ \frac{\text{Writes}_t(i-1)}{F_{S,t}(i)} & i > \text{innermost tensor level} \end{cases} \\ \text{Reads}(i) = \sum_{t\in\{W,I,O\}} B_{i,t}\text{Reads}_t(i)\end{gathered} \tag{11}$$

4.3 Latency Modeling

We calculate the latency cycles required for compute by dividing the total number of multiply-accumulate (MAC) operations in a layer by the product of all spatial factors $f_{S,i,d}$ in a mapping (i.e., the number of parallel processing elements utilized). To compute memory access latency, we divide the total number of memory accesses by the memory bandwidth. We calculate the memory latency for each memory level i utilized in Gemmini, including accumulator SRAM, scratchpad SRAM, and DRAM. We consider the maximum latency among all memory levels and the compute as the final latency since performance is limited either by memory or compute. The latency formulations are provided below:

$$\begin{gathered}\text{Compute\_Latency} = \frac{\text{\# of MACs in Layer}}{\prod_{(i,d)\in M\times D} f_{S,i,d}} \\ \text{Accesses}(i) = \text{Reads}(i) + \text{Updates}(i) + \text{Writes}(i) \\ \text{Mem\_Latency}(i) = \frac{\text{Accesses}(i)}{\text{Bandwidth}(i)} \\ \text{Mem\_Latency} = \max_{i\in M}(\text{Mem\_Latency}(i)) \\ \text{Latency} = \max(\text{Compute\_Latency}, \text{Mem\_Latency})\end{gathered} \tag{12}$$



(a) Latency: MAE=0.01% **(b) Energy: MAE=0.18%** **(c) EDP: MAE=0.18%**

Figure 4: Error of DOSA differentiable model prediction with respect to Timeloop for 100 random Gemmini configurations, 73 unique layers, 10,000 total mappings. MAE=Mean Absolute Error.

4.4 Energy Modeling

Energy is modeled via data collected for a 40nm process using Accelergy [47] and its Aladdin [39] and CACTI [29] plug-ins. In our model, compute, register access, and DRAM access energy are constant per word, whereas SRAM access energy per word scales with the number of SRAM rows and columns. The specific energy per access (EPA) values for each component are in Table 2.

$$\text{Energy}(i) = \text{Accesses}(i) \times \text{EPA}(i)$$
$$\text{Energy} = \text{MACs} \times \text{EPA}_{PE} + \sum_{i \in M} \text{Energy}(i) \tag{13}$$

4.5 Composing Performance Metrics

In this work, we target co-design of an accelerator for a given DNN model. Thus far, all calculations have been per-layer. To compute the minimal hardware requirement for a set of layers, we take the max over layers of each hardware parameter. To compute performance, for example via energy-delay product (EDP), we sum the energies and latencies of each layer, and multiply these sums at the end. For layers that appear multiple times, one mapping is generated and its energy and latency are each multiplied by the number of times the corresponding layer appears. By setting the total EDP value as the gradient descent loss term, we minimize EDP for the full model, rather than find mapping/hardware configuration that minimizes EDP for individual layers. Say a model consists of layers l.

$$\text{EDP}(model) = \left(\sum_{l \in model} \text{Energy} \right) \times \left(\sum_{l \in model} \text{Latency} \right) \tag{14}$$

Due to the scalability of gradient descent, we are able to optimize this objective with respect to all mappings in parallel, rather than one mapping at a time. This forms a different optimization problem over mappings compared to two-loop searchers that optimize for the EDP of individual layers. The flexibility of the GD loss function also enables the user to weight layers differently, which could be explored in future work.

4.6 Correlation to Timeloop

To demonstrate that our model does not compromise accuracy in order to provide differentiability and interpretability, we compare the predictions generated by the DOSA differentiable model to Timeloop [34] and Accelergy. Specifically, we evaluate 73 matrix multiplication and convolutional layers, each of which are mapped onto 100 randomly generated Gemmini configurations and sampled approximately evenly for a total of 10,000 mappings. Figure 4 shows that the EDP results from our differentiable model are on average within 0.18% of Timeloop, with 98.3% of results within 1% of Timeloop. For very small layers with very low energy usage, there is up to 12.0% error. For these very small layers, Timeloop uses a ceiling function to compute energy based on the number of blocks accessed in DRAM, whereas the DOSA differentiable model computes energy from the number of elements accessed. Apart from these small layers, high correlation is observed because DOSA captures the same relationships between mapping and latency and energy performance as Timeloop does. However, Timeloop models these relationships as an iterative program while DOSA manages to express them in a mathematical framework to enable the use of white-box optimization algorithms for DSE.

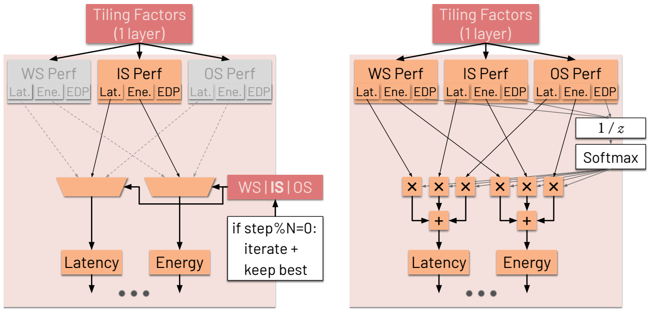
4.7 Real Hardware Performance Modeling

In general, analytical models do not completely capture hardware performance [44, 49]. Variations caused by specific implementation details and complex hardware-software interactions may be unknown to the designer or difficult to capture mathematically. One potential solution is to augment analytical models with learned surrogate models. Since many learned models, such as deep neural networks or polynomial regression models, are differentiable, DOSA is particularly well-suited to work with such models.

In this case, we train a deep neural network to predict the difference between our analytical model’s latency predictions for a layer and the real latency of Gemmini-RTL, evaluated using FireSim [16]. The model’s inputs include the layer’s dimensions, a mapping (represented as in Section 3.1.2), and a hardware configuration. The model’s architecture is similar to that of the model used in Mind

Temporal Tiling Factors	GD
Spatial Tiling Factors	GD
Spatial Tiling Dimensions	Constant
Tensor Bypass	Constant
Loop Ordering	Exhaustive

Table 5: Search algorithms for different design decisions.



(a) Iterative optimization, with IS loop ordering currently selected.

(b) Gradient-based optimization with Softmax weighting.

Figure 5: Energy and latency prediction flow different loop ordering optimization schemes.

Mappings [9]. It contains 7 hidden fully-connected layers and a total of 5737 parameters.

5 DOSA OPTIMIZATION

Constructing a differentiable performance model allows DOSA to optimize hundreds of parameters (tens per layer, times tens of layers) at once using gradient descent (GD). As seen by its use for training neural network models with up to billions of parameters, GD is a highly performant and scalable optimization method.

5.1 Search Strategy

Table 5 summarizes the search algorithms used by DOSA to explore different mapping and design decisions. To determine the temporal and spatial tiling factors for each layer in the network (with 20 variables per layer), DOSA employs GD. The GD loss term is the total performance metric, whose construction is described in Section 4.5. Differentiability is implemented using PyTorch automatic differentiation. GD start points are generated via random hardware configuration, plus CoSA [11] mappings.

We fix the spatial tiling dimensions (dataflow) decision to a weight stationary C(input channel)-K(output channel) mapping, as this is the primary spatial dataflow supported by the Gemmini generator. Note that it is possible for the dataflow decision to be incorporated into the differentiable performance model, similar to the spatial tiling factor decision, by allowing the spatial factors from all problem dimensions to exceed 1.

For the current bypassing setup, we allocate one level of buffers for tensors with different data precisions, specifically the scratchpad for inputs and weights and accumulator for outputs.

5.2 Loop Ordering

We present two potential strategies for searching loop orderings in this section, which are compared in Section 6.2.

5.2.1 *Iterative Loop Ordering Optimization.*

For the iterative optimization strategy, depicted in Figure 5a, we shuffle the order of loops for each layer every time mappings are rounded to the nearest valid mapping as explained in Section 3.1.2. This typically occurs after several hundred gradient descent steps. We select between three loop orderings per layer per level, each minimizing the data accesses for weights, outputs, and inputs respectively. We call these weight-stationary (WS), input-stationary (IS), and output-stationary (OS) orderings. The differentiable model-predicted loop ordering that minimizes overall EDP, as in Equation 14, is selected.

5.2.2 *Gradient-Based Loop Ordering Optimization with Softmax Weighting.*

The second optimization strategy involves integrating loop ordering into the gradient descent-based search by modifying the loss function. Like with iterative optimization, we consider WS, IS, and OS loop orderings for each level, for each layer. At every gradient descent step, we now consider the latency and energy of all loop ordering options and take them into account when updating tiling factors. We do so by combining the latency and energy predictions for each ordering using the softmax function $\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$. The case where loop ordering is flexible for one level is depicted in Figure 5b. We first construct a vector of energies for each loop ordering options and a similar vector of latencies.

$$\begin{aligned} \vec{\mathrm{E}_l} &= \left[\mathrm{Energy}_{l,WS} \quad \mathrm{Energy}_{l,IS} \quad \mathrm{Energy}_{l,OS}\right] \\ \vec{\mathrm{L}_l} &= \left[\mathrm{Latency}_{l,WS} \quad \mathrm{Latency}_{l,IS} \quad \mathrm{Latency}_{l,OS}\right] \end{aligned} \tag{15}$$

We then compute a vector $\vec{w_l}$ for each layer by applying the softmax function over the inverse EDPs of each loop ordering option. EDP is inverted so lower EDPs result in greater values in $\vec{w_l}$.

$$\vec{w_l} = \sigma\left(\frac{1}{\vec{\mathrm{E}_l} \odot \vec{\mathrm{L}_l}}\right) \tag{16}$$

We use $\vec{w_l}$ to weight the energy and latency values of each loop ordering option before combining the energies and latencies of all layers in the model. The new loss function is as follows:

$$\mathrm{Loss} = \left(\sum_{l \in model} \vec{w_l} \cdot \vec{\mathrm{E}_l}\right) \times \left(\sum_{l \in model} \vec{w_l} \cdot \vec{\mathrm{L}_l}\right) \tag{17}$$

With this new loss function, gradient descent passes a gradient through all paths where $\vec{w_l}$ is not equal to 0, meaning tiling factors are optimized with awareness of which loop ordering is optimal for the current tiling factors. $\vec{w_l}$ also prevents these additional gradients from hindering optimization by weighting the gradients of more performant loop orderings more heavily than gradients of less performant loop orderings.

5.3 Other Optimization Details

5.3.1 *Start Point Rejection.*

In subsequent iterations of start point generation, if a start point's differentiable model-predicted performance is more than 10× that of the best start point seen thus far, it is rejected and a new hardware configuration is selected.

Training Workloads	Target Workloads
AlexNet [20]	BERT [5]
ResNeXt-50-32x4d [51]	ResNet-50 [8]
VGG-16 [41]	RetinaNet [25]
DeepBench [30]	UNet [36]
(OCR and Face Recognition)	

Table 6: Training workloads for DNN-based performance prediction (Sections 4.7 and 6.5) and target workloads on which we evaluate DOSA (Section 6). RetinaNet performance is evaluated on layers that are not part of its ResNet backbone.

5.3.2 *Rounding.* Since gradient descent may result in non-integer tiling factors, before any mapping is evaluated, it is rounded to the nearest valid mapping. This is done by rounding each tiling factor to the nearest divisor of its corresponding problem dimension, subject to the constraint that the rounding process does not cause the product of tiling factors for that dimension to exceed the total problem size. This process iterates from the innermost to the outermost memory level.

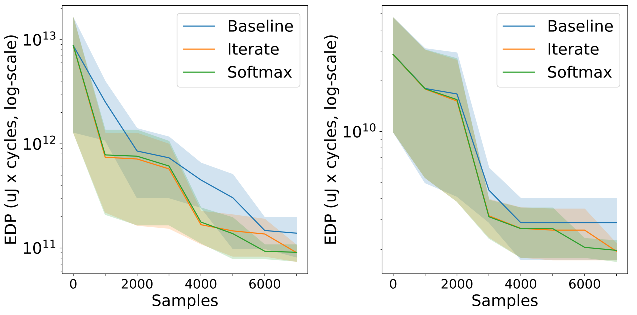
5.3.3 *Preventing Exploration of Invalid Mappings.* We do not include tiling factors at the outermost (DRAM) level as optimization targets, and instead infer them by dividing the total problem size at each dimension by the product of the rest of the tiling factors for that dimension. In order to prevent exploration of invalid tiling factors that are less than 1, a loss term is added:

$$\sum_{(k,i,d)\in\{S,T\}\times M\times D} max(1-f_{k,i,d},0) \tag{18}$$

5.3.4 *Pipeline Fusion.* Multilayer pipeline fusion is a critical technique to enhance the performance of DNN models whose computation can be decomposed into parallel streams of sequential layers. It allows concurrent processing across DNN layers, leading to higher hardware utilization and increased throughput. DOSA is able to optimize critical DSE decisions with multilayer pipelined mappings. Specifically, DOSA remains effective for determining numerical variables such as spatial/temporal tiling factors, and finding the best compute/buffer sizes (using a mapping-first approach). However, DOSA faces challenges in making discrete pipeline fusion decisions, particularly when deciding the number of layers to fuse, balancing larger intermediate buffers against recomputation, and allocating DRAM bandwidth to different layers. In this work, we do not search the space of pipeline fused layers.

6 EVALUATION

To differentiate between Timeloop performance evaluations and cycle-accurate evaluations of Gemmini RTL, from this point onward we use Gemmini-TL to refer to the Timeloop architectural definition of an accelerator analogous to Gemmini, and Gemmini-RTL to refer to the RTL implementation of Gemmini[2]. In this section, we first analyze the performance of an accelerator analogous to Gemmini using Timeloop simulation (Gemmini-TL), then demonstrate the ability of DOSA to transfer to Gemmini-RTL.



(a) ResNet-50 **(b) BERT**

Figure 6: Comparison of no loop ordering optimization by DOSA ("Baseline"), iterating over loop orderings every time mappings are rounded ("Iterate"), and gradient-based loop ordering ("Softmax"). The shaded regions represent a 95% confidence interval across 3 runs.

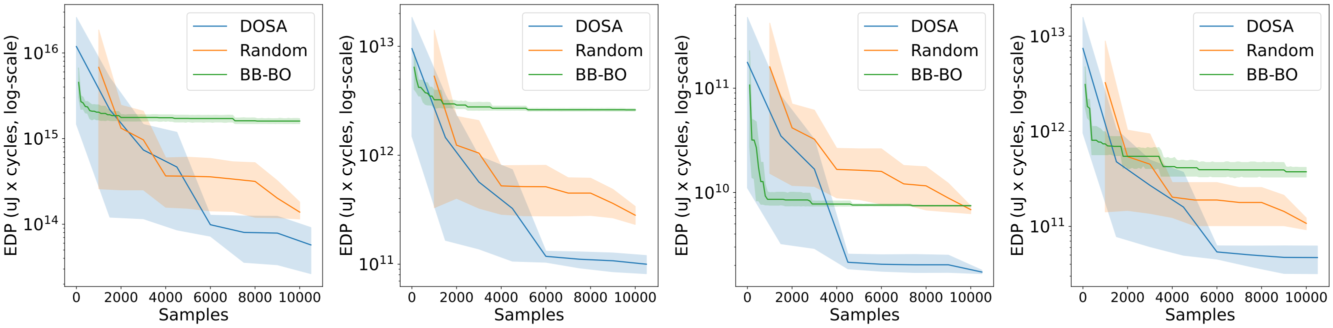
6.1 Experimental Setup

We compare the performance of DSE algorithms on a variety of target DNN models that can handle a diverse set of tasks, such as natural language processing, image classification, object detection, and image segmentation. These target models are listed in Table 6. The hardware parameters we select using the capacity requirement calculations in Section 4 are PE dimensions, accumulator SRAM sizing, and scratchpad SRAM sizing. PE dimensions come from spatial tiling factors, which can be directly used as they are always positive integers. PE array size is capped at 128x128. SRAM sizes are rounded up to increments of 1 KB. For these experiments, the specific descent algorithm DOSA uses is Adam, an optimizer similar to gradient descent with momentum. In Section 6.2, we use 7 start points, rounding happens every 300 steps, and GD is run for 890 steps on each start point. In Sections 6.3–6.5, we use 7 start points, rounding happens every 500 steps and GD is run for 1490 steps on each start point.

In addition to using CoSA to initialize GD start points, we apply it as a constant mapper to separate the effects of hardware and mapping improvements. CoSA requires a fixed partitioning for buffers that contain multiple tensors—we set up CoSA to partition the scratchpad equally between inputs and weights. Our Bayesian optimization-based hardware-mapping optimizer is a two-loop method which trains a Gaussian process model with 100 hardware designs and 100 mappings per layer per hardware design, and uses this model to select the hardware design and mappings with the best predicted performance from 1000 candidates per problem. We select these hyperparameters based on Spotlight [38]. Finally, the random search baseline evaluates 10 hardware designs with 1000 mappings per layer per hardware designs.

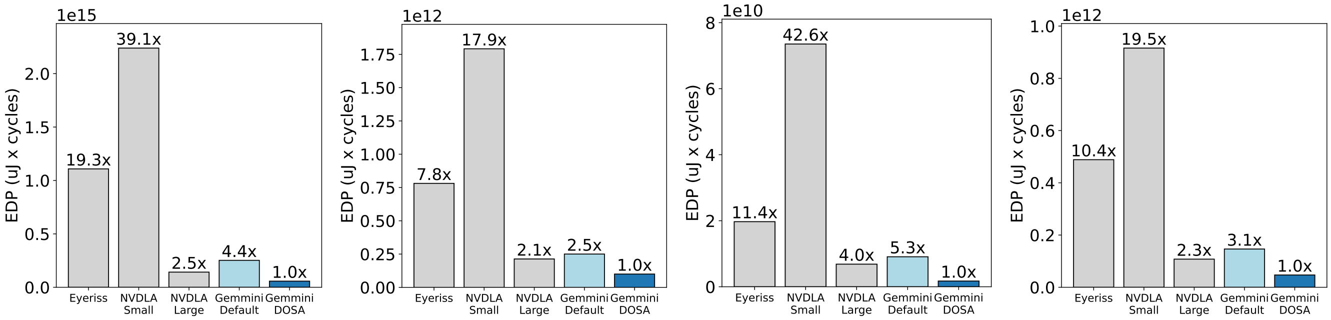
In Sections 6.2–6.4, we use Timeloop and Accelergy (with Aladdin and CACTI as plug-ins) to evaluate latency and energy. In Section 6.5 we use RTL simulation in FireSim to evaluate latency, and Timeloop and Accelergy to evaluate energy.

[2] Available at https://github.com/ucb-bar/gemmini.



(a) U-Net **(b) ResNet-50** **(c) BERT** **(d) RetinaNet**

Figure 7: DOSA EDP optimization of Gemmini-TL on 4 distinct workloads, versus baselines. Each line represents the mean (across 5 runs) best point found after x model evaluations. The shaded regions represent a 95% confidence interval across 5 runs.



(a) U-Net **(b) ResNet-50** **(c) BERT** **(d) RetinaNet**

Figure 8: Energy-delay product (EDP) of baseline accelerators, compared to DOSA-optimized Gemmini-TL. Bar labels represent EDP normalized to Gemmini-TL DOSA.

6.2 Evaluating Loop Ordering Optimization Strategies

We evaluate iterative loop ordering optimization and gradient-based loop ordering optimization on a subset of target workloads, specifically ResNet-50 and BERT. Each method uses the same start points. As shown in Figure 6, we find that gradient-based loop ordering optimization finds 1.58× better design points after 7000 samples compared to not searching loop orderings at all, whereas iterative optimization improves EDP by 1.70× over the baseline. Potential performance gains from searching loop orderings seem to be realized at similar levels by both methods, but slightly better by iteration after rounding. Iterative loop ordering optimization also requires significantly less computation. We use iterative loop ordering optimization for the experiments that follow.

6.3 Hardware-Mapping Co-Search Performance

Our evaluation finds that DOSA is able to identify significantly more performant co-design points than either random search or Bayesian optimization with a similar number of samples. BB-BO uses Timeloop simulation as a black-box optimization metric for Gemmini-TL. The random search- and DOSA-generated co-design points are also evaluated under this setup. After around 10,000 model evaluations, the geometric mean of EDP improvements for DOSA versus random search is 2.80×, and 12.59× versus BO. Evaluations done using Timeloop are considered equivalent to evaluations done using DOSA's differentiable model.

We find that in the regime of roughly less than 1000 samples, BB-BO performs best, likely because it performs more hardware search. However, only using 100 samples per hardware design may limit the full exploitation of each explore hardware design point, as by the 5,000 to 10,000 sample regime, BB-BO is overtaken by random search and DOSA. BB-BO exhibits the least variance, while random search and DOSA exhibit relatively high variance. However, DOSA does tend to converge after around 6,000 samples, perhaps as it reaches close to optimal EDP.

In Figure 8, we compare Gemmini-TL, with hardware and mappings optimized by DOSA, with other expert-optimized accelerator baselines (Eyeriss, NVDLA Small, NVDLA Large, and Gemmini default) for the four target workloads. We evaluate these accelerators using Timeloop and search 10,000 valid mappings per layer using the Timeloop random-pruned mapper. Gemmini-TL configurations searched by DOSA consistently outperform the baselines by more than 2× in EDP.

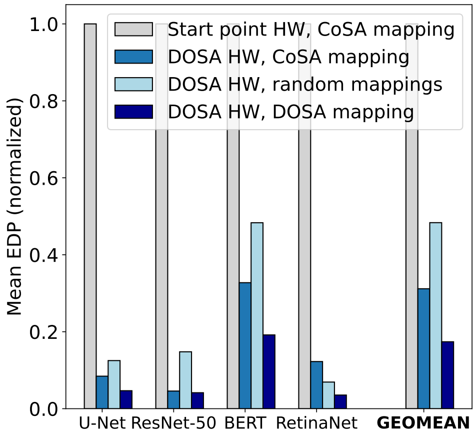


Figure 9: DOSA improves performance under a constant mapper and produces near-optimal mappings for the hardware design points it selects.

6.4 Separating the Effects of Hardware and Mapping Search

We further find that DOSA identifies both performant hardware design points and performant mappings. We run gradient descent 10 times, and use Timeloop to compare EDP at the GD start point (randomly selected hardware design with CoSA mappings) to EDP at the GD end point (DOSA-generated hardware designs and mappings). We also evaluate these DOSA-generated hardware designs with CoSA mappings, to see whether the hardware design improves under a constant mapper. This case study shows that DOSA produces a 5.75× improvement over start point performance (geomean over 4 workloads, 10 GD instances each). Furthermore, with CoSA as a constant mapper, DOSA generated hardware designs show a 3.21× improvement. This shows that end point hardware designs are better than start point hardware designs and that the performance improvements gained by DOSA are not simply due to the use of a performant mapper (CoSA) in the loop.

DOSA mappings also show a 1.79× improvement over CoSA and a 2.78x improvement over a 1000-sample random mapper on the same hardware, showing that DOSA identifies near-optimal mappings, competitive with or beating a state-of-the-art mapper, on the hardware design points it generates. The results per workload are shown in .

6.5 Gemmini-RTL Optimization with DOSA

In this section, we assess the efficacy of our one-loop differentiable-model-based gradient descent approach for real hardware design. We explore three potential approaches for latency modeling of Gemmini-RTL: an *analytical-only* approach using the model described in Sections 4.1–4.5, a *DNN-only* model trained from scratch from measured Gemmini-RTL performance data and our *DNN-augmented analytical model* approach from Section 4.7. Note that energy is predicted using the DOSA differentiable analytical model in all cases.

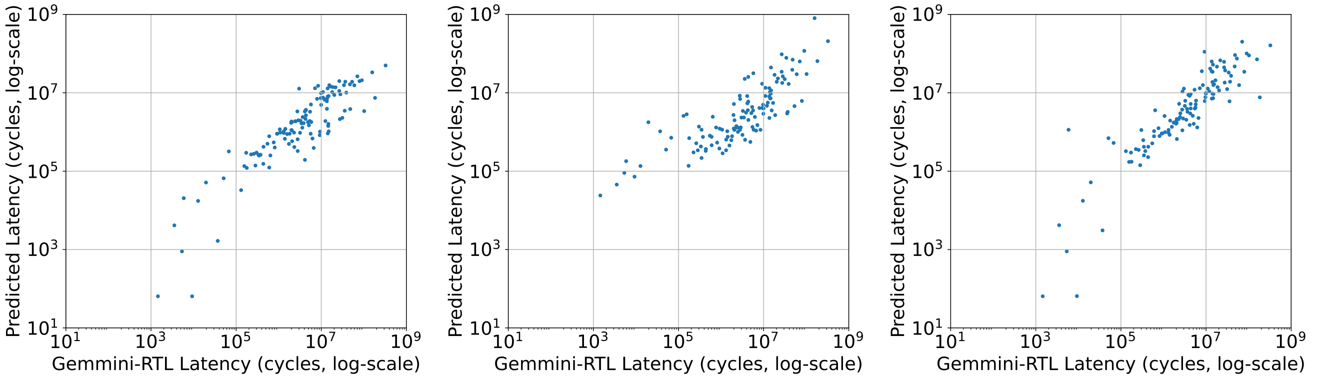
6.5.1 DNN Model Training Setup. We utilize FireSim to generate cycle-accurate Gemmini-RTL latencies on our training set of models (Table 6). Specifically, we generate a relatively small dataset of 1567 random mappings, roughly evenly distributed among the layers in Table 6. This dataset is used to train the two DNN performance models, which have the same architecture and are trained using the same hyperparameters. The models are trained for 50,000 epochs. The models could likely be trained to greater accuracy with a larger dataset, but we limit dataset size to more accurately represent a real hardware design environment.

6.5.2 Prediction Accuracy. Figure 10 compares the accuracy of each approach when modeling Gemmini-RTL performance on unseen random mappings of workloads in the training set. To quantify prediction accuracy, we measure the Spearman rank correlation of each model's predictions. Spearman rank correlation measures the strength of the associations between two variables by assessing the monotonicity of the relationship between those variables [45]. The combined model providing the highest accuracy, with a correlation coefficient of 0.92. The analytical-only model is the next most accurate model on this dataset with a correlation coefficient of 0.87, and the DNN-only model is the least accurate by a small margin with a correlation coefficient of 0.84.

6.5.3 Optimization Performance. To evaluate DOSA's real hardware optimization performance, we run DOSA using the analytical-only, DNN-only and DNN-augmented latency models. Specifically, for each latency model and for each target workload, we generate a predicted optimal set of mappings and buffer sizes for 16x16 PE Gemmini-RTL, fixing PE dimensions and adjusting only buffer sizing and mappings. We fix PE dimensions to keep accelerator size in the same order of magnitude as default Gemmini-RTL. This allows for a more apples-to-apples comparison against the hand-tuned default design point, and ensures that generated configs can be simulated using FireSim. We compare the performance of DOSA-generated mappings to the mappings generated by the Gemmini-RTL default heuristic-based mapper, and the default scratchpad and accumulator sizings of 128 KB and 32 KB respectively (256 KB and 64 KB including double-buffering), which were selected using heuristics similar to those in Interstellar [52]. As mentioned above, Gemmini-RTL latency is evaluated using FireSim, while energy is evaluated using Timeloop and Accelergy, with CACTI as a plug-in.

Figure 11 shows the prediction accuracy of each model on the mappings produced using DOSA during these experiments. These are mappings for layers not present in the training dataset. On this dataset, the DNN-only model is clearly less accurate than the analytical-only model or the DNN-augmented analytical model, as seen by the outliers in Figure 11b. This reflects previous work [14], which has shown that DNN-only methods for mapspace exploration [9] have difficulty generalizing beyond their training sets. The analytical model, which is not fit to any particular workload, actually improves in prediction accuracy compared to the original test dataset (Figure 10a), likely because this new dataset consists of performant mappings generated by DOSA, and as such is more uniform.

Figure 12 shows the EDP of Gemmini-RTL after it is optimized for each target workloads, using the three latency models. The analytical-only and DNN-only models respectively yield 1.48× and

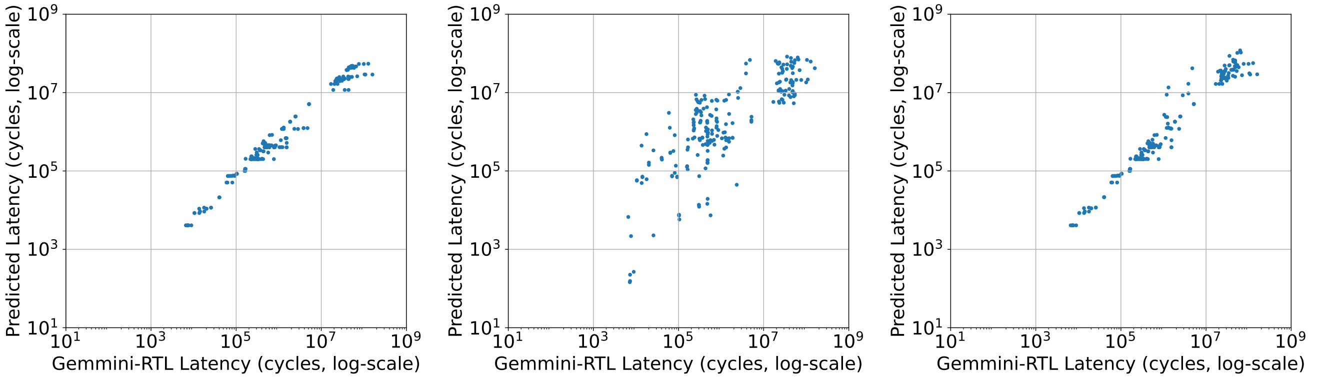


(a) Analytical Only, corr.=0.87

(b) DNN Only, corr.=0.84

(c) Analytical + DNN, corr.=0.92

Figure 10: Accuracy of Gemmini-RTL latency models on test split of random mappings (training workloads from Table 6, unseen mappings). Correlation metric is Spearman rank correlation.



(a) Analytical Only, corr.=0.97

(b) DNN Only, corr.=0.79

(c) Analytical + DNN, corr.=0.97

Figure 11: Accuracy of Gemmini-RTL latency modeling on mappings generated using DOSA. These are mappings for the target workloads in Table 6, which are not included in DNN training data.

1.66× improvements over Gemmini's default buffer sizes and tilings. Despite its drop-off in *prediction accuracy*, the DNN-only model outperforms the analytical model in *optimization performance* on the three workloads other than U-Net. U-Net contains weight sizes unseen in the training set (Table 6), again demonstrating DNN models' difficulty in generalizing.

When DNN and analytical models are combined, we improve optimization performance even further, to 1.82× over default, while maintaining a level of prediction accuracy (on DOSA-generated points) higher than that of the DNN-only model and similar to that of the analytical-only model. Unlike the DNN-only model, the DNN-augmented analytical model does not produce outliers in our experiments, since its outputs are constrained using the analytical model prediction.

Table 7 shows the buffer sizes selected by DOSA with the DNN-augmented latency model. The buffer size ratios (scratchpad size divided by accumulator size) identified by DOSA using the analytical+DNN model setup range from 1.28 to the original heuristically selected ratio of 4. We find that for all four target workloads, DOSA sizes both the accumulator and scratchpad significantly larger than the default sizes, indicating that these buffers may be underprovisioned for such workloads. Furthermore, for the three convolutional neural networks (U-Net, ResNet-50, and RetinaNet), the ratio of scratchpad to accumulator size is smaller than in the default configuration.

This experiment demonstrates that DOSA's gradient descent-based optimization technique is compatible with performance models other than the analytical model presented in this work. In fact, DOSA allows for a more iterative process when moving from simulation to real hardware, as the model for each objective (latency, energy, and in future work, potentially area) can be replaced and augmented independently as demonstrated here. Furthermore, the

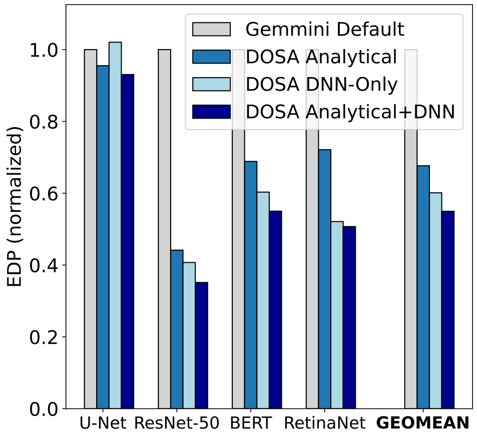


Figure 12: DOSA optimization of Gemmini-RTL using various performance models, compared to Gemmini’s hand-tuned hardware and mapper.

		Accumulator (KB)	Scratchpad (KB)
Gemmini Default		32	128
DOSA-Optimized Gemmini-RTL	U-Net	123	322
	ResNet-50	196	251
	BERT	64	256
	RetinaNet	112	261

Table 7: Gemmini configurations generated by DOSA Analytical+DNN.

components that remain analytically modeled can be easily tuned as more accurate data becomes available or design decisions change. For example, energy-per-access numbers could be updated based on measured numbers once a process node is selected or modified, which would be orders of magnitude more efficient than generating large amounts of data for a DNN model to consume.

7 CONCLUSION

In this paper, we present DOSA, a model-based approach mapping-first DSE. By constructing a differentiable analytical performance model for a DNN accelerator, we can use gradient descent to perform an efficient one-loop co-search of both the hardware and mapping spaces. This enables us to to perform DSE targeting multi-layer neural net workloads, attaining an EDP 2.80× better than random search and 12.59× better than Bayesian optimization, while using a similar number of samples. Furthermore, we find that DOSA not only improves hardware design performance by 3.21× under a constant mapper, but also beats a state-of-the-art mapper on the hardware designs it selects.

DOSA demonstrates that interpretable, designer-trusted architectural modeling and ML-based optimization methods can be combined to improve the convergence of DSE. We pair our analytical latency model with a DNN model trained on RTL simulation data, improving EDP over the default Gemmini configuration 1.48× with just the analytical model and by 1.82× when analytical and learned models are combined. With this work, we move one step closer to bridging the gap between architectural models and real silicon.

ACKNOWLEDGMENTS

Many thanks to the anonymous MICRO reviewers and our shepherd for suggesting important additions. We also thank Hasan Genc and Divija Hasteer for their help with Gemmini, and Xiangyu (Sam) Xu and Jonathan Wang for their help generating baseline data.

This research is supported by NSF Award CCF-2238346 and by SLICE Lab industrial sponsors and affiliates Amazon, AMD, Apple, Google, Intel, and Qualcomm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Z. Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: a system for Large-Scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[2] Amazon. 2018. AWS Inferentia: High Performance Machine Learning Inference Chip. https://aws.amazon.com/machine-learning/inferentia/.

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIXSymposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[6] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. arXiv:1406.2661 [stat.ML]

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.

[10] Qijing Huang, Charles Hong, John Wawrzynek, Mahesh Subedar, and Yakun Sophia Shao. 2022. Learning A Continuous and Reconstructible Latent Space for Hardware Accelerator Design. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[11] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham,

Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[13] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.

[14] Sheng-Chun Kao, Angshuman Parashar, Po-An Tsai, and Tushar Krishna. 2022. Demystifying Map Space Exploration for NPUs. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 269–281. https://doi.org/10.1109/IISWC55918.2022.00031

[15] Sheng-Chun Kao, Michael Pellauer, Angshuman Parashar, and Tushar Krishna. 2022. DiGamma: domain-aware genetic algorithm for HW-mapping co-optimization for DNN accelerators. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 232–237.

[16] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 29–42. https://doi.org/10.1109/ISCA.2018.00014

[17] John Karro, Greg Kochanski, and Daniel Golovin. 2017. Black box optimization via a bayesian-optimized genetic algorithm. *Proc. OPTML* (2017), 10th.

[18] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A learned performance model for tensor processing units. In *Proceedings of Machine Learning and Systems (MLSys)*.

[19] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[21] Aviral Kumar, Amir Yazdanbakhsh, Milad Hashemi, Kevin Swersky, and Sergey Levine. 2021. Data-Driven Offline Optimization for Architecting Hardware Accelerators. In *Workshop on ML for Systems at the Conference on Neural Information Processing Systems (NeurIPS)*.

[22] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[23] Gary Lauterbach. 2021. The path to successful wafer-scale integration: the Cerebras story. *IEEE Micro* 41, 6 (2021), 52–57.

[24] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. 2021. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.

[25] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*. 2980–2988.

[26] Yujun Lin, Mengtian Yang, and Song Han. 2021. NAAS: Neural Accelerator Architecture Search. In *Design Automation Conference (DAC)*.

[27] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. Tenet: A framework for modeling tensor dataflow based on relation-centric notation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[28] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. 2021. ZigZag: Enlarging joint architecture-mapping design space exploration for DNN accelerators. *IEEE Trans. Comput.* 70, 8 (2021).

[29] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman Jouppi. 2009. Cacti 6.0: A tool to model large caches. *HP Laboratories* (01 2009).

[30] Sharan Narang and Greg Diamos. 2017. Baidu DeepBench. *GitHub Repository* (2017). http://www.github.com/baidu-research/DeepBench

[31] NVIDIA. 2018. *TensorRT: https://developer.nvidia.com/tensorrt*.

[32] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU:. https://www.nvidia.com/en-us/data-center/a100/.

[33] OpenAI. 2020. ChatGPT. https://openai.com/blog/chat-gpt-3-launched/. Accessed: April 25, 2023.

[34] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer, 234–241.

[37] Amit Sabne. 2020. Xla: Compiling machine learning for peak performance. (2020).

[38] Chirag Sakhuja, Zhan Shi, and Calvin Lin. 2023. Leveraging Domain Information for the Efficient Automated Design of Deep Learning Accelerators. In *International Symposium on High-Performance Computer Architectural (HPCA)*. IEEE.

[39] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 97–108. https://doi.org/10.1109/ISCA.2014.6853196

[40] Zhan Shi, Chirag Sakhuja, Milad Hashemi, Kevin Swersky, and Calvin Lin. 2020. Using Bayesian Optimization for Hardware/Software Co-Design of Neural Accelerators. In *Workshop on ML for Systems at the Conference on Neural Information Processing Systems (NeurIPS)*.

[41] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[43] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J. Dally, Joel Emer, Stephen W. Keckler, and Brucek Khailany. 2019. Magnet: A modular accelerator generator for neural networks. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.

[44] Irene Wang, Prasenjit Chakraborty, Zi Yu Xue, and Yen Fu Lin. 2022. Evaluation of gem5 for performance modeling of ARM Cortex-R based embedded SoCs. *Microprocessors and Microsystems* 93 (2022), 104599. https://doi.org/10.1016/j.micpro.2022.104599

[45] Eric W. Weisstein. [n. d.]. Spearman Rank Correlation Coefficient. ([n. d.]). https://mathworld.wolfram.com/SpearmanRankCorrelationCoefficient.html

[46] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[47] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942149

[48] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. 2022. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE.

[49] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2015. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 577–589. https://doi.org/10.1109/HPCA.2015.7056064

[50] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. Hasco: Towards agile hardware and software co-design for tensor computation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[51] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*.

[52] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.

[53] Amir Yazdanbakhsh, Christof Angermueller, Berkin Akin, Yanqi Zhou, Albin Jones, Milad Hashemi, Kevin Swersky, Satrajit Chatterjee, Ravi Narayanaswami, and James Laudon. 2021. Apollo: Transferable Architecture Exploration. *arXiv preprint arXiv:2102.01723* (2021).

[54] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A Full-Stack Search Technique for Domain Optimized Deep Learning Accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 27–42. https://doi.org/10.1145/3503222.3507767

A ARTIFACT

A.1 Abstract

This section describes how to access the artifacts for DOSA, and run several of the experiments from Section 6, in particular reproducing the results of Gemmini optimization using DOSA. We reproduce experiments in both Timeloop and RTL simulation environments.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** User machine, AWS FPGA Developer AMI 1.12.2.
- **Hardware:** AWS EC2 instances (c5.4xlarge, f1.2xlarge).
- **Metrics:** Accelerator energy-delay product. Evaluated on Timeloop architectural simulator and FireSim RTL simulation.
- **Output:** Plots showing model accuracy, DSE sample efficiency, optimization performance. Energy and latency numbers for Gemmini hardware configurations and mappings.
- **Experiments:** DSE sample efficiency and EDP of resulting designs compared to baselines.
- **How much disk space required:** 20 GB (on user machine), 200 GB (on EC2 instance).
- **How much time is needed to prepare workflow:** 2 hrs.
- **How much time is needed to complete experiments:** approximately 24 hrs, dependent on user machine.
- **Publicly available:** Yes.
- **Code licenses:** Multiple, see download.
- **Archived:** https://doi.org/10.5281/zenodo.8358253

A.3 Description

A.3.1 How to access. The artifact includes several git repositories archived on Zenodo at https://doi.org/10.5281/zenodo.8358253.

- The code for DOSA, stored in the archive `dosa.zip`, plus a version with all submodules initialized under `dosa-full.zip`.
- The FireSim, Chipyard, and Gemmini configurations (hardware and software) used for DOSA, archived under the names `firesim-dosa.zip`, `chipyard-dosa.zip`, and `gemmini-dosa.zip` respectively.

A.3.2 Hardware dependencies. One AWS EC2 c5.4xlarge instance ("manager" instance), and one f1.2xlarge instance ("run farm" instance) are required. The latter will be launched automatically by FireSim's manager. We have provided a pre-built FPGA image to avoid the long latency (about 8 hours) of the FPGA build process.

A.3.3 Software dependencies. DOSA, Timeloop, and Accelergy can be installed on most Linux-based user machines.

For FireSim-based experiments, use ssh or mosh on your local machine to remote access EC2 instances. All other requirements are automatically installed by scripts in the following sections.

A.4 Installation

A.4.1 Installing DOSA. On a user machine with Python 3.10 or greater, clone the archived DOSA code:

```
$ curl -Ls -w %{url_effective} -o a https://doi.
      org/10.5281/zenodo.8358253 > DL_url
$ wget $(cat DL_url)/files/dosa.zip
$ unzip dosa.zip
```

First, acquire a Gurobi optimizer license[3] and download it to path of choice (`$license_path`). Next, run the following:

```
$ export GRB_LICENSE_FILE=($license_path)
$ cd dosa
$ pip3 install -e .
```

A.4.2 Timeloop and Accelergy. Install Timeloop and Accelergy on the user machine. The following dependencies are required (command provided for Debian-based systems):

```
$ sudo apt install scons libconfig++-dev libboost
      -dev libboost-iostreams-dev libboost-
      serialization-dev libyaml-cpp-dev
      libncurses-dev libtinfo-dev libgpm-dev git
      build-essential python3-pip
```

Timeloop and Accelergy are available as submodules of the DOSA repository. First, install Accelergy and its plug-ins. Within `dosa`:

```
$ git submodule update --init --recursive
$ cd accelergy-timeloop-infrastructure/src/
      accelergy
$ pip3 install .
$ cd ../cacti
$ make
$ cd ..
$ mv cacti ~/.local/bin/
$ cd ../accelergy-cacti-plug-in
$ pip3 install .
$ cd ../accelergy-aladdin-plug-in
$ pip3 install .
$ cd ../accelergy-table-based-plug-ins
$ pip3 install .
$ accelergy
$ accelergyTables
```

Install Timeloop and add its executables to your `PATH`:

```
$ cd ../timeloop/src
$ ln -s ../pat-public/src/pat .
$ cd ..
$ scons --accelergy --static -j4
$ export PATH=$PATH:$(pwd)/build
```

The Timeloop[4] and Accelergy[5] documentation may be helpful if any issues arise.

A.4.3 FireSim-Based Experiments. First, follow the instructions on the FireSim website [6] to create an EC2 manager instance. Complete the steps in the "AWS EC2 F1 Getting Started Guide". Once you have completed up to and including "Setting up your Manager Instance / Key setup, Part 2" in the FireSim docs, you should have a manager instance set up, with an IP address and key. Use ssh or mosh to log in to the instance. Next, in `/home/centos`, clone the archived FireSim repository.

[3] https://www.gurobi.com/features/academic-named-user-license/
[4] https://timeloop.csail.mit.edu/timeloop/installation
[5] https://timeloop.csail.mit.edu/accelergy/installation
[6] https://docs.fires.im/en/1.17.1/Getting-Started-Guides/AWS-EC2-F1-Getting-Started/index.html

```
$ curl -Ls -w %{url_effective} -o a https://doi.
      org/10.5281/zenodo.8358253 > DL_url
$ wget $(cat DL_url)/files/firesim-dosa.zip
$ unzip firesim-dosa.zip
```

Run the following, which will initialize dependencies and set up FireSim and Chipyard:

```
$ cd firesim-dosa
$ git checkout dosa
$ ./build-setup.sh
$ sudo yum install autoconf
$ source sourceme-f1-manager.sh
$ firesim managerinit --platform f1
```

After sourcing, complete the steps in "Setting up your Manager Instance / Completing Setup Using the Manager". Note that `sourceme-f1-manager.sh` must be sourced every time you log in to the instance.

Finally, get the FPGA image used for this experiment. Go to `firesim-dosa/deploy`, and within `config_hwdb.yaml` paste the contents of the file in `built-hwdb-entries/` (there should be one file containing a YAML-formatted entry).

A.5 Evaluation and expected results

A.5.1 Figure 4: Analytical model correlation with Timeloop. On the **user machine**, run the following commands. This will correlate DOSA's differentiable model against Timeloop for our 10,000 point dataset and store the error plots to `output_dir/error_<metric>.png`.

```
$ cd dosa
$ ./fig4.sh
```

A.5.2 Figure 7: Optimization of Gemmini-TL versus baseline algorithms. In the same environment, run the following script, selecting one workload:

```
$ ./fig7.sh (unet|resnet50|bert|retinanet)
```

This will take several hours to run, per workload, and generate a plot at `output_dir/network_searcher_<workload>_log_<timestamp>.png`. This corresponds to the plot to Figure 5, but over one run rather than averaged over 5. Results should fall within or close to the confidence bounds of the original plot.

A.5.3 Figure 8: Comparison to hand-tuned accelerators. Only after running `fig7.sh` for the corresponding workload, run:

```
$ ./fig8.sh (unet|resnet50|bert|retinanet)
```

The plots will be generated at the location `output_dir/arch_compare_<workload>_<timestamp>.png`. Since these are based on the results of one run rather than averaged over 5, results here will again vary slightly compared to the original plot.

A.5.4 Figures 10 and 11: Gemmini-RTL performance prediction accuracy. Run the following script:

```
$ ./fig10.sh
```

This will reproduce the plots in Figures 10 and 11 under `output_dir/predict_<predictor>_<dataset>.png`. These plots show the prediction accuracy of the three different predictors on the two datasets of Gemmini-RTL latency, which were previously generated using FireSim.

A.5.5 Figure 12: Optimization of Gemmini-RTL. Now, **move to the AWS EC2 instance** set up with the FireSim fork. To run the full workflow of Figure 12, we would need to train two DNN models, run DOSA (constraining the number of PEs to 16x16), select the mappings with the best predicted performance, evaluate latency with FireSim, then combine with energy numbers from Accelergy. To reduce runtime and work that must be done across both the user machine and EC2 instance, we provide the mappings generated by DOSA during this experiment directly to the evaluator as part of our FireSim fork. To build the software for a given workload and run FireSim, run the following:

```
$ cd ~/firesim-dosa/target-design/chipyard/
      generators/gemmini/software/gemmini-rocc-
      tests
$ ./artifact_script.sh (analytical|both|dnn) (
      unet|resnet50|bert|retinanet)
```

The first argument to `artifact_script.sh` indicates which of the three latency predictors from the previous section should be used. The second argument indicates the target workload. This script launches FireSim automatically and should take a few minutes to run. Depending on the target workload, FireSim will generate either one or two directories under `deploy/results-workload`, for matrix multiplication and/or convolutional layers. Pass the previously selected options, along with the directories (`($result_dir_1)`, and potentially `($result_dir_2)`) to the parsing script.

```
$ cd ~/firesim-dosa/target-design/chipyard/
      generators/gemmini/software/gemmini-rocc-
      tests
$ python parse_results.py
      --pred (analytical|both|dnn)
      --workload (unet|resnet50|bert|retinanet)
      --result ($result_dir_1)
      --result ($result_dir_2)
```

This will update the CSV file located at `gemmini-rocc-tests/artifact/<predictor>/<workload>.csv`. **Copy this file back to the user machine**, to your choice of path (`$workload_csv`). On the user machine, run the following to print out the EDP of the Gemmini default mapper/HW and the EDP of the mappings/HW found by DOSA, all using latency numbers from FireSim. The relative magnitude of the Gemmini default and DOSA EDPs should match those in Figure 12.

```
$ ./fig12.sh (unet|resnet50|bert|retinanet) (
      $workload_csv)
```

When you are done evaluating, go to the EC2 console and terminate your instance(s).

A.6 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html