

Demystifying Map Space Exploration for NPUs

Sheng-Chun Kao¹ Angshuman Parashar² Po-An Tsai² and Tushar Krishna¹

¹Georgia Institute of Technology

²Nvidia

¹skao6@gatech.edu, tushar@ece.gatech.edu

²{aparashar, poant}@nvidia.com

Abstract

Map Space Exploration is the problem of finding optimized mappings of a Deep Neural Network (DNN) model on an accelerator. It is known to be extremely computationally expensive, and there has been active research looking at both heuristics and learning-based methods to make the problem computationally tractable. However, while there are dozens of mappers out there (all empirically claiming to find better mappings than others), the research community lacks systematic insights on how different search techniques navigate the map-space and how different mapping axes contribute to the accelerator’s performance and efficiency. Such insights are crucial to developing mapping frameworks for emerging DNNs that are increasingly irregular (due to neural architecture search) and sparse, making the corresponding map spaces much more complex. In this work, rather than proposing yet another mapper, we do a first-of-its-kind apples-to-apples comparison of search techniques leveraged by different mappers. Next, we extract the learnings from our study and propose two new techniques that can augment existing mappers — warm-start and sparsity-aware — that demonstrate speedups, scalability, and robustness across diverse DNN models¹.

1. Introduction

Deep Neural Network (DNNs) have become an indispensable tool in the solution toolbox for a variety of complex problems such as object detection, machine translation, language understanding, autonomous driving, and so on. There is growing demand for specialized DNN accelerators (also called Neural Processing Units or NPUs)² pursuing high performance with high energy, power, and area efficiency.

The performance and energy-efficiency of a NPU depends on how a DNN is *mapped* over the accelerator’s hardware (compute and memory) resources [35, 44]. Specifically, a mapping (*aka* schedule) includes the computation order, parallelization strategy and tile sizes [35, 44], as shown in Fig. 1. In order to achieve high efficiency across a wide range of DNNs that include diverse layer shapes and sizes, state-of-the-art DNN accelerators are often designed with

flexibility to support different mapping strategies [9, 36, 48]. This flexibility imposes a unique challenge for deployment: finding a high-quality mapping between a DNN and the flexible accelerator from the space of all legal mappings (i.e., the *map space*) during compile time. This is crucial to unlock the full potential of the DNN accelerator.

As a result, prior work has clearly defined *map space exploration* (MSE) [19, 23, 28, 44], as a critical problem for NPU design and/or deployment, cleanly separating it from the hardware architecture design space exploration (DSE) problem. DSE includes identifying the right compute and memory configurations for the NPU within constraints such as total FLOPS, area, and power. MSE, meanwhile, takes the hardware configuration and DNN workload as input and finds optimized mappings, optimizing some objective (e.g., latency or energy-efficiency). To perform MSE, various search algorithms (i.e., *mappers*) have been proposed within the past few years [2, 3, 7, 12–15, 23, 25, 41, 44, 49, 50, 54, 55, 57–60, 63, 64, 66, 67, 70, 73, 75, 76, 79].

Despite the success achieved by these prior efforts, MSE remains a computationally challenging problem. This is because the search space for legal mappings for even a single layer of a modern DNN (e.g., ResNet-50) on a typical edge class accelerator [9] is $\sim O(10^{24})$ [19, 28] which would require more time than the age of the earth to search exhaustively (assuming 1ms to evaluate each mapping sample). This gets exacerbated as newer and ever larger DNN models are being created with increasing frequency, especially thanks to the success of neural architecture search techniques [4, 5, 39, 47, 61]. Furthermore, the advent of *compressed-sparse* DNNs [16, 38, 40, 51, 68, 69, 80], whose mappings are not performance-portable across sparsity levels (a key finding in this paper), further increases MSE burden.

Researching more sophisticated scalable and sparsity-aware MSE techniques is at least partially hampered by the fact that even though prior approaches have *empirically shown* that their techniques work, none of them demonstrate *why* they work and the insight behind their optimization techniques.

It is these very insights that we wish to extract in this paper, and in the process demystify MSE as a problem. We cover both heuristics and learning-based optimization approaches, analyze their behavior, and learn from their best traits. We then use these learnings to scale MSE to more complex workloads.

1. Code available at <https://github.com/maestro-project/gamma-timeloop>.

2. In this paper, we use the terms DNN Accelerator and NPU interchangeably.

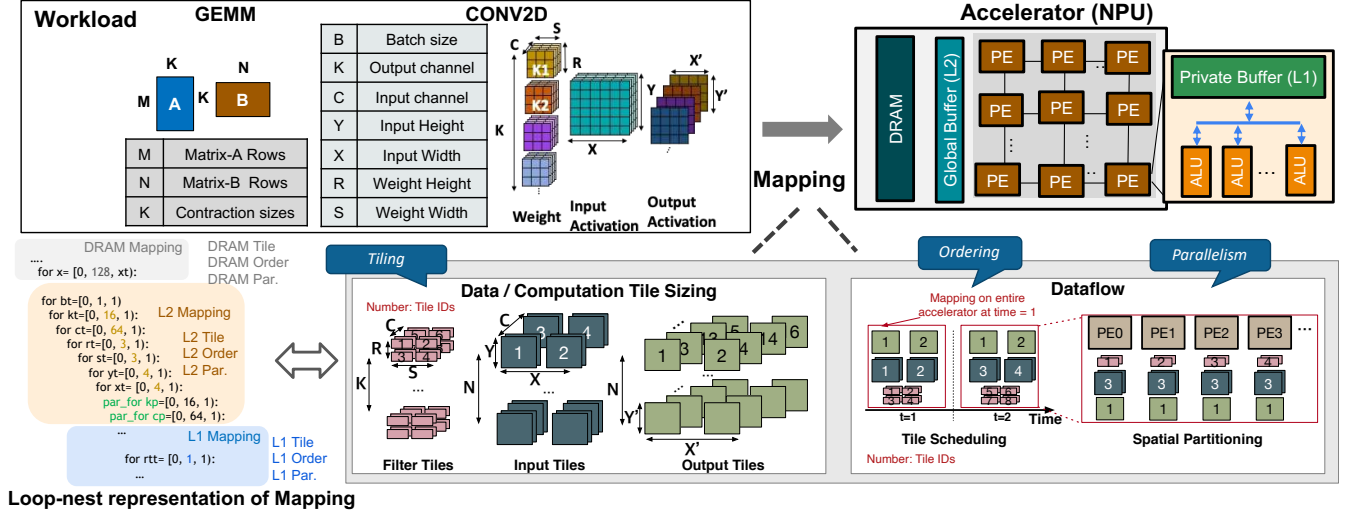


Fig. 1: The overview of DNN Workload, Accelerator, and a (NVDLA-like [1]) Mapping.

Specifically, our contributions are two-fold.

(1) This is the first work, to the best of our knowledge, to *quantitatively* compare three wide categories of mappers: random-based [44] (i.e., heuristic pruning), feedback-based [28] (i.e., blackbox optimization and reinforcement learning), and gradient-based [19] (i.e., surrogate models), and analyze their trade-offs. We conduct a sensitivity analysis of different mapping axes to understand the contribution of each axis. We then perform case studies that reveal distinguishing characteristics of good and bad mappings. Our analysis reveals that: (i) random search is inefficient, (ii) gradient-based search converges fast but requires prior knowledge of the accelerator architecture, and (ii) feedback-based search is more adaptable and sample-efficient, but requires higher cost to acquire each sample. Our analysis also shows that optimality of a dense DNN mapping does not port over to a sparse DNN.

(2) Based on our findings, we propose two novel heuristic techniques to advance the state-of-the-art in MSE: (i) We propose a *warm-start* technique to initialize the MSE with prior optimal solutions from previous layers in a replay buffer based on a *similarity* metric, enabling the mapper to start at a better point and converge faster. In our evaluations, we find that warm-start can help the mapper converge to a similar performance point 3.3x-7.3x faster. (ii) We also propose a *sparsity-aware* technique to search for a mapping that can perform well across a range of target activation sparsities. A fixed mapping found by our sparsity-aware approach can achieve 99.7% of the performance of each of the mappings specifically tailored to the various density levels.

2. Background: DNN Accelerators

2.1. DNN Workloads

In this work, we use individual DNN layers/operators as our target *workload*. The workloads vary across different DNN models because of different types of operations such as

CONV2D, Depth-wise CONV, Point-wise CONV, Attention, Fully-Connected (FC), and so on, and different tensor shapes for the layers (i.e., batch, input, weight kernel sizes), as shown in Fig. 1. All these operations can be represented with a loop-nest of computations. For example, a CONV2D can be represented as 7 for-loops, and GEMM can be represented as 3 for-loops.

2.2. Accelerator Hardware Configuration

A canonical NPU often houses a spatial array of Processing Elements (PEs), as shown in Fig. 1. Each PE has one to several ALU units to compute partial sums, and private local (aka “L1”) buffers to store weights, input activations and partial sums. The accelerator also houses a global shared (aka “L2”) buffer to prefetch activations and weights from DRAM for the next tile of computation that will be mapped over the PEs and L1 buffers. Networks-on-Chip are used to distribute operands from the global L2 buffer to the L1 buffers in the PEs, collect the partial or full outputs, and write them back to the L2 buffer.

2.3. Accelerator Map-Space

Given a DNN workload, there exist several choices for *mapping* it on the accelerator’s PEs and buffer hierarchy over space and time. The mapping includes the following components [34, 44], shown in Fig. 1:

(1) **Tile sizes:** The ability to change bounds and aspect ratios of data tiles from one or more operand tensors per level of the buffer hierarchy [46].

(2) **Loop order:** The ability to change the loop orders iterated per tiling level.

(3) **Loop parallelization:** The ability to change which tensor dimensions are parallelized per tiling level. This represents the *spatial* partitioning of data (i.e., across PEs).

Fig. 1 shows an example of the mapping used by the NVDLA [1] accelerator. Choices for (2) and (3) together are often referred to as *dataflow* [34] which has been

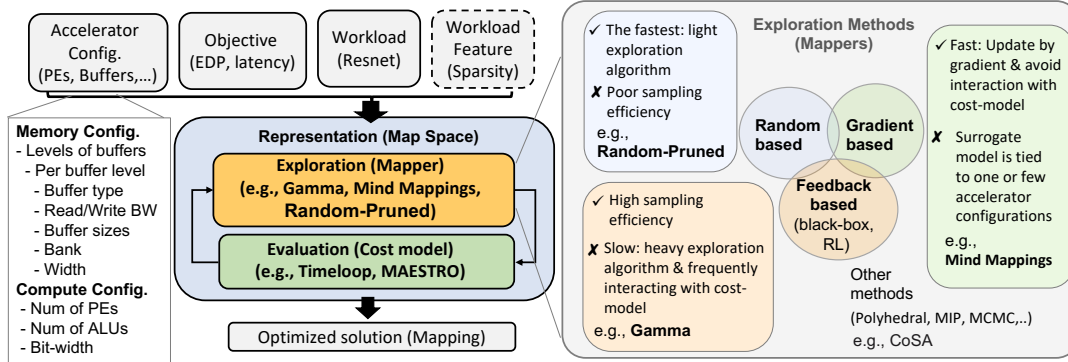


Fig. 2: A canonical Map Space Exploration framework.

informally classified by prior work into weight-stationary, output stationary and input-stationary [8]. The design-space of all possible mappings (i.e., dataflows + tile-sizes) that an accelerator can support is called its *Map-Space* [44].

Flexible DNN accelerators [9, 36] allow a mapping optimizer within a compiler to explore tile sizes, loop orders and parallelization independently for each layer. This mapping flexibility is crucial for accelerators to adapt to growing diversity in DNNs [34]. The overall runtime and energy-efficiency of an accelerator depends on both the hardware configuration and the mapping, making it crucial to find an optimized mapping³, [34, 44, 75], as we discuss next.

3. Map Space Exploration (MSE)

A canonical MSE framework is shown in Fig. 2. MSE takes the NPU’s HW configuration (§2.2) and target DNN workloads (size, shape, and additional features such as sparsity level of weight and/or activations) as input and finds optimized mappings given an objective (e.g., latency, throughput, energy, energy-delay-product (EDP), and so on). MSE may be run at compile time within a mapping optimizer [6] after the NPU is deployed, or at design-time in conjunction with DSE for co-optimizing the mapping and HW configuration [31, 73].

The MSE process often includes three parts: *Representation* of search space, *Evaluation method*, and *Exploration method*. The representation will define the scope of the searching problem and the size of the search space. An optimization loop that includes exploration and evaluation performs the actual search. The optimization continues till the MSE converges, or reaches a given sampling budget or wall-clock run time budget.

3.1. Representation of Map Space

While recent work has proposed various representations (MAESTRO [35], UNION [24], and Ruby [22]) to increase mapping diversity in the map space, in this work we leverage the canonical Timeloop representation, which is loop-nests

3. In this paper, we focus on finding optimized mapping for individual DNN layers/operators, which has been the target of most Map-Space Exploration tools. We leave Inter-layer mappings via operator-fusion as future work.

to represent each tiling level (e.g., NVDLA-like mapping in Fig. 1). We ensure that all the candidate mappings generated by various mappers during MSE are legal.

3.2. Evaluation Method (Cost Model)

MSE relies on a DNN accelerator *cost model* to estimate the performance of a certain mapping on a given accelerator for a given workload. These cost models are typically analytical, enabling rapid evaluation of different design-points in a matter of ms. Some widely used cost models include Timeloop [44], MAESTRO [34], dMazeRunner [12], Interstellar [75], SCALE-sim [52] and others [32, 42]. These cost models can model different kinds of accelerators (systolic arrays [52], flexible spatial arrays [12, 34, 44], sparse accelerators [71], and so on) and capture each accelerator’s map space in different formats. In this work, we use Timeloop [44] as our cost model⁴ which is validated against real chips [10, 54].

3.3. Exploration Method (Mapper)

The exploration algorithm in MSE (Fig. 2) is called a mapper. Dozens of different DNN mappers have been proposed, which we categorize into *random search based* [12, 44, 54, 63, 75], *feedback-based* (including reinforcement learning and black-box optimization) [7, 25, 27, 28, 73, 79], *gradient-based* [19], and *others* (including mathematical optimization, MCMC, polyhedral transformations, and heuristics) [3, 15, 23, 25, 49, 64] (Fig. 2). The random search-based either apply random sampling on the search space or apply pruned random search [6, 44], which prunes off the redundant search space to increase the sampling efficiency. The feedback-based use a learning algorithm to interact with the cost model and keep improving its solution. The run time of both random search-based and feedback-based depend heavily on the run time of the cost model, potentially becoming the bottleneck of the MSE run time. Gradient-based methods uses a *differentiable* surrogate model, which eliminates this bottleneck and can update the solution directly by the gradient of the loss. We do a deeper dive within these three types in §4.3.

4. Timeloop includes *both* a cost model and mappers. Throughout this paper, we refer to the former as Timeloop and the latter as Timeloop-mapper. Timeloop-mapper itself supports a variety of search heuristics, with the default being Random-Pruned which we use. We also run other mappers using Timeloop as the cost model.

3.4. Why MSE Matters

MSE bridges the gap between two active trends: (1) efficient DNN model design [11, 53, 62] (which has led to a huge diversity in layer shapes/sizes and emergence of sparsity in state-of-the-art DNN models) and (2) flexible hardware accelerators that support diverse mappings (dataflows + tile sizes) via configurable buffer hierarchies [46] and on-chip interconnect topologies [36, 48] as an answer to the first trend. MSE is crucial for extracting performance and energy-efficiency from the accelerator as there can be multiple orders of difference in performance and energy-efficiency between good and bad mappings, as prior works have demonstrated [19, 28, 44].

While several mappers are being actively developed [2, 3, 7, 12–15, 23, 25, 41, 44, 49, 50, 54, 55, 57–60, 63, 64, 66, 67, 70, 73, 75, 76, 79], there is no work, to the best of our knowledge, that has focused on understanding how different mappers navigate the map-space, how different mapping axes contribute to the performance, and trade-offs between search approaches, which is the focus of this work.

4. Quantitative MSE Analysis

In this section, we perform a quantitative analysis of the three classes of mappers described in §3.3 to identify *when* and *why* one works better than the other. The goal of this analysis is to educate the DNN accelerator research community on Mapper design, rather than propose yet another mapper.

4.1. Methodology

Workload. We consider workloads from different models: Resnet [18], VGG [56], Mnasnet [61], Mobilenet [53], and Bert-large [65]. Some frequently referenced workloads across different experiments are described in Table 1.

Hardware Accelerator. We model the NPU using Timeloop [44]. We assume three-levels of buffer hierarchies: DRAM, a 64KB shared global buffer, and 256B private local buffer for each of the 256 PE. Each PE houses 4 ALU units (Accel-B in Table 1). We also model the NPU the Mind Mappings paper [19] uses (Accel-A), whose configuration is similar but with different sizing as shown in Table 1.

For analyzing sparse mappings (§4.5), we use TimeloopV2, *aka Sparseloop* [71, 72], as the cost model to explore the map space in a flexible sparse accelerator, and leverage Gamma as the mapper. Besides tiling, ordering and parallelism, Sparseloop also models hardware and software optimizations (e.g., power gating and compressed tensors) in sparse DNN accelerators.

Objective. We use multi-objective – Energy and Latency (Delay), throughout the optimization process. When optimization finishes, we select the solution with the highest Energy-Delay-Product (EDP) on the Pareto frontier. We use EDP as the performance criteria of found mapping. Note that any formulation of the objective can also be used such as power, area, performance-per-watt, performance-per-mm², and so on.

TABLE 1: The description of the relevant workloads and accelerator configurations used across evaluations.

Workload	(B,K,C,Y,X,R,S)		Accelerator Configuration
Resnet Conv_3	(16,128,128,28,28,3,3)	Accel A	512 KB shared buffer, 64 KB private buffer per PE, 256 PEs, 1 ALUs per PE
Resnet Conv_4	(16,256,256,14,14,3,3)		
Inception Conv_2	(16,192,192,27,27,5,5)		
Workload	(B,M,K,N)		
Bert-Large KQV	(16,1024,1024,512)	Accel B	64 KB shared buffer, 256 B private buffer per PE, 256 PEs, 4 ALUs per PE
Bert-Large Attn	(16,512,1024,512)		
Bert-Large FF	(16,4096,1024,512)		

Experiment Platform. We run experiments using a desktop with a 12-core Intel I7-6800K CPU and a Nvidia GTX1080 to train the surrogate model in Mind Mappings.

4.2. Size of Map Space

The size of the map space heavily depends on representation. In this paper, we follow the efficient representation used by Timeloop to represent the three mapping axes. We use CONV2D (7 for-loop) as workload and 3-level of buffer hierarchy (DRAM, L2, L1) as architecture configuration as an example to guide the discussion of map space.

Tile sizes. Buffers at each level of the scratchpad memory hierarchy will have a dedicated tile size for each of the dimensions, as shown by the different tile sizes within the 7 for-loops of the L2 mapping in Fig. 1. The total possible combination depends on the tensor shape of each workload and increases exponentially with the number of buffer hierarchies.

Loop Order. Each buffer level would have a dedicated permutation of loop order. E.g., in Fig. 1, the loop order in L2 mapping from outer to inner loop is (B,K,C,R,S,Y,X). The total combinations become $(7!)^3$ (we have 3 buffer levels in our example).

Parallelism. Parallelism happens across levels of compute units (2-level of compute units in Fig. 1, i.e., across PEs and ALUs). At each level of the compute unit, we can choose to parallelize from 0 (no parallelism) to 7 (all parallelism) dimensions. The total combination becomes $2^{7 \times 2}$.

Map-Space. The Cartesian product of these sub-spaces leads to the size of the entire map space, which is at the level of $O(10^{21})$ for the workloads discussed in §4.1.

4.3. Understanding Mapper Sampling Efficiency

Recall from §3.3 that we categorize state-of-the-art mappers into three major techniques (Fig. 2). We select state-of-the-art mappers out of each category - Timeloop’s Random-Pruned [44] from random-based, Gamma [28] from feedback-based, and Mind Mappings [19] from gradient-based methods⁵. - and compare their characteristics with respect to search speed and sampling efficiency⁶.

5. Random-Pruned and Mind Mappings both natively work with the Timeloop cost model. Gamma was originally demonstrated with MAESTRO, and we extended it to use the Timeloop cost model. We leave the task of porting representative mappers from the *others* category (§3.3 to a common cost model and analyzing them as future work.

6. The performance improvement over number of sampled points.

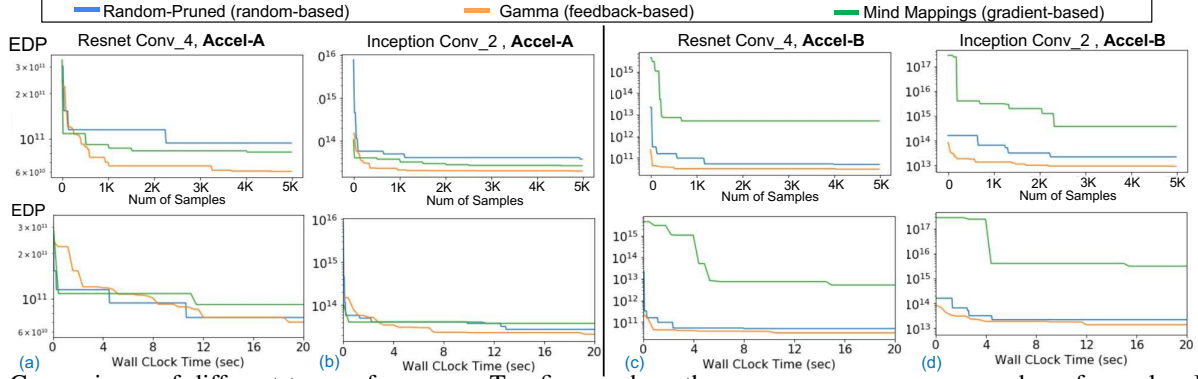


Fig. 3: Comparisons of different types of mappers. Top figures show the converge curve across number of samples. Bottom figures show the converge curve across wall clock time.

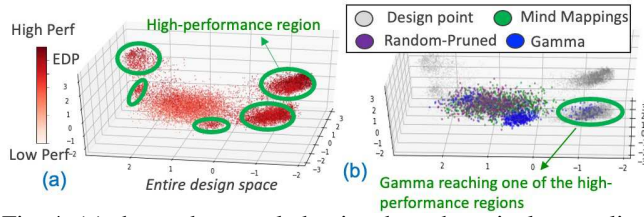


Fig. 4: (a) shows the sampled points by exhaustively sampling the search space of (Resnet Conv_4, Accel-A). The 3D visualization is projected by PCA dimension reduction. (b) shows the sampled points of different types of mappers in this search space.

- **Random-Pruned (random-based):** Random-Pruned [44] uses random sampling on a pruned search space. The pruning strategies are based on heuristics, e.g., permutations do not matter for the innermost tiling level and for tile sizes that are one [44].
- **Gamma (feedback-based):** Gamma [28], a genetic algorithm (GA) based method, keeps a population of candidate solutions, uses specifically designed mutation operators to perturb populations to explore different mapping axes (tile, order, parallelism), and uses crossover to create next generations of populations. Gamma has been shown to beat other optimization techniques, including reinforcement learning [28, 30].
- **Mind Mappings (gradient-based):** Mind Mappings [19] trains a neural-network-based surrogate model via offline sampling of millions of data points collected from the cost model. It uses the loss gradient to update its solution. During MSE, it utilizes gradient-descent on this surrogate model to find mappings, instead of searching.

In the following evaluation case study, we show two sets of NPU configurations (Table 1): *Accel-A*, on which the surrogate model is trained for MindMappings, and *Accel-B*, an unseen accelerator configuration for the surrogate model.

4.3.1. Trained Accelerator Configuration (Accel-A). Iso-sampling points Comparisons. We set the sampling budget to 5,000 points and compare the sampling efficiency of

algorithms in the top figures of Fig. 3(a)(b). The random-based method progresses the slowest over number of samples. Among the gradient-based and feedback-based, the gradient-based method progresses faster at the start owing to its direct gradient feedback. However, with more number of samples, the feedback-based method starts to perform better. It is because the gradient-based method is more prone to fall into local optimum (discussed later) while the feedback-based methods typically work well for global optimization problems.

Iso-time Comparisons. We set a tight time budget, 20 seconds, and track the performance to wall clock time in the bottom figures of Fig. 3(a)(b). Despite their better sampling efficiency, the feedback-based and gradient-based methods do not show a clear edge over the random-based method within tight wall-clock run time budget. Random-based methods do not have costly built-in learning algorithms as the other two and hence can run more number of samples given the same time budget, which is essential when the run time budget is strictly tight. Specifically, the run time of the searching algorithm in Gamma and Mind Mappings is about 10x larger than Random-Pruned.

4.3.2. Accelerator configuration not in the Training Dataset (Accel-B). We use the same set of workloads as in Fig. 3(a)(b), but change the accelerator configuration to Accel-B, which is not in the training dataset of the surrogate model of the gradient-based method. As shown in Fig. 3(c)(d), the gradient-based method cannot perform as well as it did for the trained accelerator configuration, Accel-A. It demonstrates that the trained surrogate model does not generalize across accelerator configurations. Note that we can also re-train the surrogate model for the new accelerator configuration, which will recover the performance. However, it will require another full-fledged DNN training. Besides, we also need to collect 1 - 5 million of new training data to achieve quality results [19].

Variance of Accelerator Configurations. The random-based and feedback-based method take workloads and accelerator configurations as inputs and therefore are agnostic to variance in accelerator configurations. In contrast, the gradient-based method train its surrogate model based on

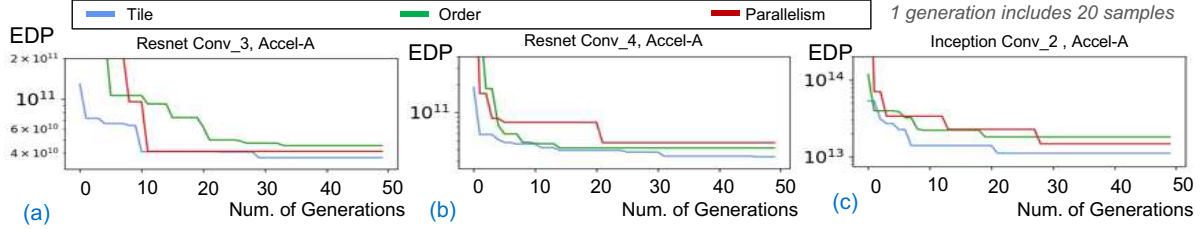


Fig. 5: Mapping axes sensitivity analysis using the mutation operators in Gamma [28]. E.g., Tile (blue): means mutating tile only, i.e., only tile is explored, and other mapping axes are fixed, similarly for (mutate-)Order and (mutate-)Parallelism.

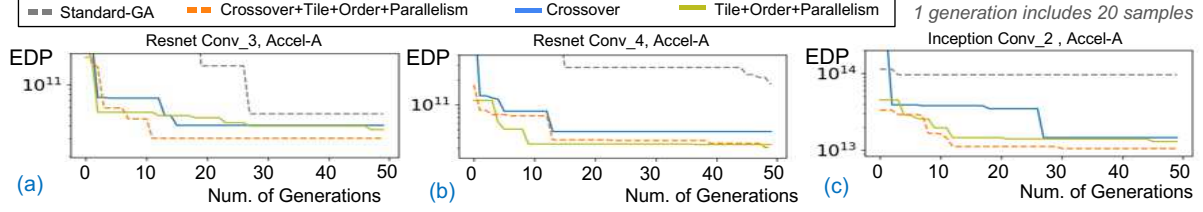


Fig. 6: Crossover (blending two mappings) sensitivity analysis using operators in Gamma [28]. Standard-GA uses the standard mutation and crossover (without domain-specific operators along each mapping axes designed in Gamma [28]).

a collected training dataset. The training dataset includes collected workloads and collected accelerator configurations. While surrogate model can generalize the workload encoding across different DNNs models [19], the generalization of accelerator configurations is more challenging since arbitrary buffer levels, buffer sizes, PE sizes, and other details (Fig. 2) can be made. Thus the surrogate model is tied to one or few accelerator configurations.

4.3.3. Visualization of the Sampling Points. To better understand how different algorithms behave in the map space, we plot their sampling points in Fig. 4 using the workload and accelerator configuration in Fig. 3(a). Fig. 4(a) shows the entire map space while dark red represent higher-performance points. There is a large low-performing region at the center while some small clusters of the high-performing points (green circle) scatter across the space. Fig. 4(b) shows the points different algorithms actually sampled. Given the limited 5,000 sampling budget, The Random-Pruned method only samples around the lower-performing region because most of the design points sit here. Mind Mappings starts with the lower-performing region and gradient-updates to the higher-performing regions at the right. However, it sits at the local optimum. Gamma also starts with a lower-performing region but can explore a wider region faster because of its population-based method (which is common in many feedback-based algorithms [17, 20, 21, 33]). Gamma reached one of the high-performance regions, as shown in Fig. 4(b).

Takeaway of comparing different mappers:

- Learning-based methods, including gradient-based and feedback-based, can keep improving the quality of the sampling function over searching iterations, leading to better sampling efficiency.
- When the time constraint is strictly tight so that the learning-based methods cannot yet gather adequate data to improve their sampling function (i.e., still at explo-

ration phase instead of exploitation), the random-based method is the most cost-effective choice.

- The surrogate model of the gradient-based method is trained on a collected training dataset, where the accelerator configuration is often fixed. The trained surrogate model cannot generalize across different accelerator configurations.

We pick Gamma, the feedback-based method, as our main mapper for the rest of the discussion in this paper.

4.4. Understanding Mapper Search Operators

Recall that there are three mapping axes in the map space, tile, order, and parallelism. Gamma has dedicated genetic operators to explore along these axes, i.e., *mutate-tile*, *mutate-order*, and *mutate-parallelism*. It also houses a *crossover* operator to blend two high-performant mappings to create the next candidate mapping samples. Note that each genetic operator is specifically tuned to adapt to this map space as shown in the Gamma paper [28], which is the key source of sampling efficiency over other black-box optimizers, including RL and standard GA. As Fig. 6 shows, full-fledged Gamma (dotted orange line) performs an order of magnitude better than standard GA across the three evaluated workloads.

4.4.1. Mapping Axis Sensitivity Analysis. In Fig. 5, we explore each mapping axis individually (keeping the other two fixed) via the mutation operator in Gamma [28] such as *mutate-tile* for tile exploration, *mutate-order* for order exploration and so on. We find *mutate-tile* to have the highest impact on EDP compared to the other components.

4.4.2. Crossover Sensitivity Analysis. Gamma has crossover operator which blends two mapping points to create the next candidate mapping points. We execute a sensitivity analysis of crossover in Fig. 6. We find that

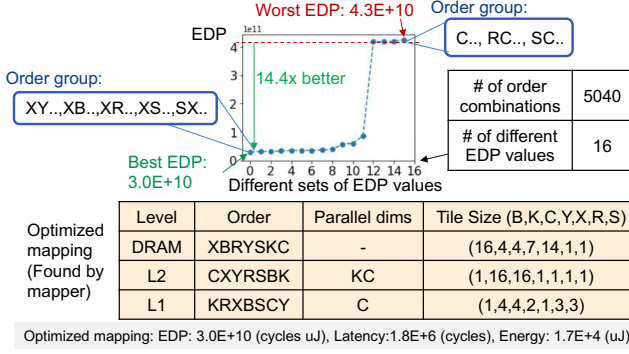


Fig. 7: The EDP difference of the same mapping with different loop order. We sweep through all $7!$ order combinations assuming all the buffer level utilize the same order. The $7!$ different mapping leads to 16 different EDP performance, with the best and the worst EDP differs by 14.4x times (under Resnet Conv_4, Accel-B).

disabling crossover (light green) can hugely impact the potential performance compared to full-fledged Gamma (dotted orange). However, crossover-only without other operators (dark blue) is also not adequate. Crossover working with all the dedicated mutation operators for the three maxing axes (dotted orange) can maximize the sampling efficiency of the mapper (Gamma) and ends up giving the most optimized performance.

Takeaway of comparing operators in a mapper:

- If one were to incrementally implement different exploration functions along the mapping axes, starting with the tile exploration would be the most cost-effective option.
- Blending two high-performance mappings (crossover) can effectively create another high-performance mapping.
- The ability to explore different order and parallelism dimensions choices is not as critical as tile size exploration to optimize EDP performance.
- Note that even when fixing the order or parallelism throughout the optimization process, at the initialization stage, we still randomly initialized order and parallelism for the initial populations (a groups of initial sampling points). It implies that few explorations of order and parallelism are often adequate to give competitive mapping. It is owing to the fact that many combinations of order or parallelism will lead to similar latency or energy performance, as we discuss later in §4.4.3.
- The performance difference of two mapping for the same problem can be as large as 3 orders of magnitude difference, consistent with prior works [19, 28, 34, 44].

4.4.3. Loop Order Sensitivity Analysis. We perform a sweep of loop order permutations to demonstrate our observation that *many order permutations lead to similar performance* as observed above. We use the found mapping in the experiment setting in Fig. 6(a) and swap out the order permutation by enumerating through all the possibilities. The search space is as large as $(7!)^3 = 1.28E+11$. We add a constraint that each level of the buffer will use the same order

TABLE 2: MSE for workload with weight sparsity. In each columns, the blue cell shows the performance of the optimized mapping for the sparse workload; the rest of the cells shows the performance of the same mapping tested with the workload with different sparsity. We highlight the best-performing cell of each row by green text. We can observe that the blue cells overlap with green texts, indicating that different workload with different sparsity levels do require different mapping to optimize the performance.

		EDP (cycles uJ)			
		Weight Density of the Workload			
Test the found mapping across different density	Density	1.0	0.5	0.1	0.01
	Density	Resnet Conv_3			
	1.0	3.7E+10	3.9E+10	5.8E+10	1.6E+12
	0.5	1.0E+10	4.9E+09	9.1E+09	3.9E+11
	0.1	8.0E+08	6.6E+07	6.4E+07	8.3E+08
	0.01	5.0E+07	3.1E+04	4.8E+04	1.6E+04
	Density	Resnet Conv_4			
	1.0	3.1E+10	3.6E+10	1.0E+11	4.3E+11
	0.5	8.3E+09	4.9E+09	1.4E+10	9.6E+10
	0.1	5.5E+08	9.1E+07	2.3E+07	3.7E+08
	0.01	3.0E+07	7.0E+05	6.4E+03	5.4E+03
	Density	Inception Conv_2			
	1.0	1.1E+13	1.3E+13	1.5E+13	5.9E+14
	0.5	3.4E+12	2.0E+12	2.3E+12	1.5E+14
	0.1	3.5E+11	1.3E+10	5.1E+09	4.0E+10
	0.01	3.3E+09	9.4E+06	3.3E+06	6.2E+05

TABLE 3: The optimized EDP performance of inner and outer product style mapping on sparse-dense GEMM workloads in Bert-large model [65]. The workload density indicates the density of the sparse matrix. Bert-large KQV: the key/ query/ value projection operations. Bert-large Attn: the attention operation, Bert-large FC: the FC operations at the end of attention blocks.

		EDP (cycles uJ)					
		Bert-large KQV		Bert-large Attn		Bert-large FC	
Workload	Density	Inner Product	Outer Product	Inner Product	Outer Product	Inner Product	Outer Product
1.0		7.6E+11	9.8E+11	1.9E+11	2.5E+11	7.8E+14	9.1E+14
0.5		1.1E+11	1.4E+11	2.8E+10	3.6E+10	1.5E+14	1.5E+14
0.1		9.0E+08	1.6E+05	3.4E+08	3.6E+08	1.4E+12	1.1E+08
0.01		1.9E+05	1.6E+05	2.0E+05	8.0E+04	1.8E+08	1.1E+08

to relax the complexity, which becomes $7! = 5,040$ choices. Fig. 7 shows that there are only 16 different EDP values out of 5,040 different mappings. We can observe some patterns in each of the same performance mapping groups, as shown in Fig. 7. For example, “XY..” means the permutation starting with XY. The loop order at the DRAM buffer level of the original mapping found by Gamma (XB..) also falls in the high-performance order group.

Takeaway. Many order permutations will lead to similar energy or latency performance. This is why various loop orders can be placed into large “stationarity” buckets (such as weight/ input/ output/ row) [8, 34, 44] or inner/ outer product [71].

4.5. Understanding Sparse Accelerator Mappings

4.5.1. Need of MSE for Flexible Sparse Accelerator.

There is a series of research proposing ways to prune DNN models [16, 38, 40, 51, 68, 69, 80]. However, the pruned models often cannot achieve as much performance gain in hardware as proven by the algorithmic analysis because of the increase complexity to find efficient mapping. There are several sparse accelerators [26, 29, 37, 45, 48, 74, 77, 78] for efficiently running sparse workloads, skipping zeros in the weights and/or activations. However, they often employ a fixed mapping (or a limited set of mappings). Given the nascent domain, MSE for flexible sparse accelerators is relatively unexplored, with one study looking into it [71] in contrast to several MSE studies for flexible dense accelerators [3, 7, 12, 15, 19, 23, 25, 27, 28, 49, 64, 73, 79]. This leaves MSE for sparse accelerators and workloads an area with plenty of opportunity to explore.

4.5.2. Mapping Search for Sparse Weights. For model pruning, we often focus on pruning out the weight of the models, essentially some weight becomes zero. Density 1.0 means dense weight, and density 0.5 means 50% of the weights are zero. In Table 2, we use workloads with different weight densities and use MSE to search for optimized mappings. The performance of found mappings are recorded in the blue cell. For example, the mapping found for Resnet CONV_3 with 0.5 density has EDP performance of 4.9E+9 (cycles uJ).

Do we need different mappings for different sparsity?

We take the optimized mapping targeting a specific workload with a specific density (blue cell) and test it with the same workload with different densities. For e.g., at the top-left blue cell (Table 2), we have an optimized mapping for the dense workload (density 1.0). Then we use the same mapping and test its performance under 0.5, 0.1, 0.01 density degrees, whose performance is recorded in the bottom cells. We perform the same experiment for the other three columns. We mark the best-performing cell across each row with green text. We can observe that the best-performing ones always located in the blue cell, meaning to optimize mapping for specific sparsity of the workload is needed to pursue the best performance. **Takeaway.** A dense mapping cannot generalize across sparsity workloads. Different sparsity levels of the workload require different mappings to maximize the performance.

4.5.3. Sparse Inner and Outer Product. An observation that many sparse accelerators papers have made is that inner product accelerators often perform better for low sparsity workloads and outer product accelerators perform better at high amounts of sparsity [43, 45]. We study this general observation using the MSE framework. We assume the underlying sparse accelerator is flexible to support both inner and outer product style mapping. Inner and outer products are essentially affecting the loop order. Therefore, we fix the loop order and perform MSE for the other two axes (parallelism and tile sizes). Table 3 shows that the inner product style with optimized mapping consistently

outperforms the outer product counterparts for workload density larger than 0.5, while the outer product style has an edge over the inner product style at densities smaller than 0.1. **Takeaway.** From the viewpoint of MSE, we are able to validate the observation that inner product style mappings are better for denser workloads while outer product style works better at high sparsity.

4.6. Lessons Learnt

We summarize two key takeaways from our analysis:

- The feedback based mapper has the highest sampling efficiency and can directly work for any workload and accelerator configurations. However, it has the highest wall-clock time to acquire one sample (10x more costly than random-based mappers, e.g., Random-Pruned [44]). Neural architecture search is leading to new DNN models coming out frequently with highly irregular tensor shapes, increasing the demand for sample-efficient MSE.
- MSE needs to consider sparsity. While the sparsity of the weight is often fixed for a trained DNN models, the sparsity of activations is dynamic. When facing activation sparsity, we would either under-utilize the hardware because of inefficient mapping or would need to re-launch the MSE again and again for every input-activation.

5. Improving MSE

From our analysis and takeaways from §4, we focus on the two open-challenges identified above for next-generation mappers: search speed and sparsity. We propose two heuristics - “warm start” and “sparsity-aware” to address these.

5.1. Warm-start

5.1.1. Motivation. We introduce **warm-start** to reduce the search time. This method is inspired by two observations. (1) Informed by the study in §4.4 and §4.4.3, we know that order and parallelism are often less sensitive from workload to workload. (2) Because of the nature of the DNN operations (CONV, FC, and others), consecutive layers often have some dimensions the same or similar to each other. Therefore potentially the mapping of the later layers can be inspired by the found mapping of the previous layer.

5.1.2. Proposed Warm-start Search Mechanism. Fig. 8 shows our warm-start flow. We introduce a *replay buffer* within the MSE framework which stores the optimized mapping of each workload (i.e., DNN layer) that has been run so far. We initialize the algorithm with the solution of the highest-similarity workload in the replay buffer.

MSE Flow. Warm-start works via the following flow. *Step-1:* When the new workload comes, we compare the workload *similarity* to the workloads in the replay buffer. We use *editing distance* as the similarity metric. *Step-2:* Initialize

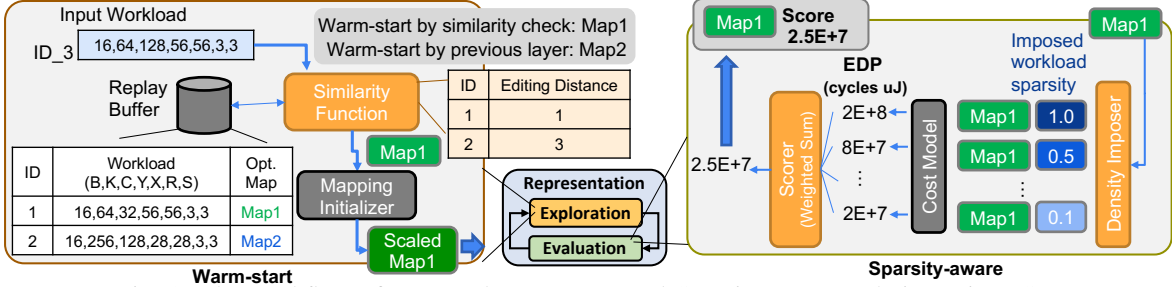


Fig. 8: The workflow of proposed Warm-start and Sparsity-aware techniques in MSE.

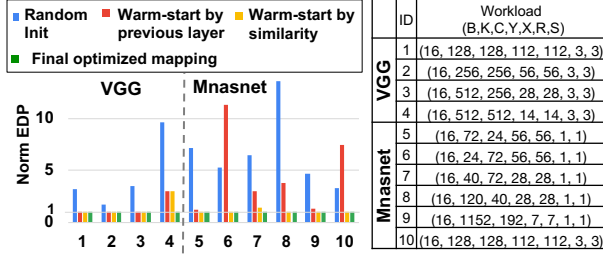


Fig. 9: Performance comparisons of initialized solution by Random Init and two types of warm-start Init comparing to the final optimized performance (after search). The EDP values are normalized by final optimized EDP (green bars). In geomean, 5.0x higher, 2.3x higher, 1.2x higher, compared to 1.0x.

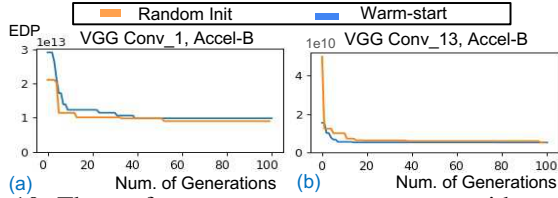


Fig. 10: The performance convergence curve with random initialization and warm-start (by similarity) initialization at the (a) first layer and (b) a later layer of VGG16.

the algorithm with the mapping with the highest-similarity by (i) Inherit the order and parallelism parts of the solution, and (ii) Scale the tile sizes to match the tensor dimensions of the current workload. *Step-3*: Run the search algorithm.

Walk-Through Example. In Fig. 8 as an example, there are two workloads that are finished with their final optimized mapping stored in the replay buffer. The next workload, workload-3, comes and will go through warm-start block before entering optimization loop. In the warm-start block, we use *editing distance* to compare the similarity between the current workload and the workloads in the replay buffer. E.g., workload-3 is only differ from workload-1 in the C-dimension, leading to editing distance of 1; similarity, editing distance with workload-2 is 3 (K, Y, X). Therefore, we pick the stored optimized mapping for workload-1 (Map1), scale it to match the tensor shape of workload-3 (i.e., multiply C tile size by 2 at the outer-most tiling level (L3 mapping)), and use it as the initialized mapping for the optimization.

Similarity. Typically, for most DNNs we find that previous layer has the highest-similarity score. However, there are some exceptions: 1) the layers can come out-of-order because

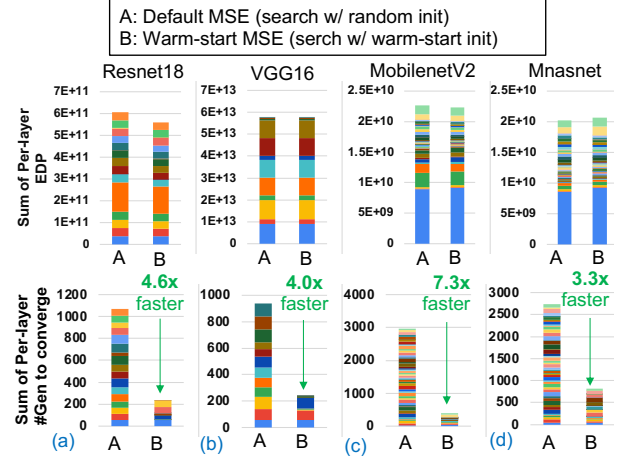


Fig. 11: The benefit of warm-start (by similarity) when executing MSE. Warm-start MSE achieves comparable EDP performance to default MSE, but converges 3.3-7.3x faster. Different colors represent different layers of the DNN models.

of other compiler decisions or 2) irregular tensor shapes of the workloads created by neural architecture search.

5.1.3. Evaluation. Impact of Warm-start Initialization.

Warm-start is an initialization technique. In Fig. 9, we show the performance of the initialized mapping of warm-start by similarity (yellow bar), warm-start by previous layers (red bar), and the default random initialization (blue bar). We evaluate workloads from two DNN models, VGG [56] and Mnasnet [61]. Many DNN models are made by human experts, where the shape of each layer are often designed with high regularity such as VGG [56] and Resnet [18]. In these models, warm-start by previous layers and warm-start by similarity make no difference, since the highest-similarity layers are almost always the previous layers, as shown in workload ID 1 - 4. However, the shape of the workloads in the Mnasnet, a network found by neural architecture search, are more irregular. Therefore warm-start by similarity becomes essential, providing 2x better performance than warm-start by previous layers. However, both warm-start strategies are effective and are 2.1x and 4.3x better than random initialization.

Impact of Warm-start Search. Warm-start reduces the time to converge. Fig. 10 shows the converge curve of the first layer and a later layer to perform MSE on VGG16 [56]. For the first layers (VGG Conv_1), there are no previous

solution in the replay buffer. Therefore, searching with random initialization or with warm-start initialization has no difference. However, for the later layers (VGG Conv_13), searching with warm-start initialized with better points and converges faster.

We perform MSE for all layers in 4 DNN models with and without warm-start. Fig. 11(a) shows that searching with warm-start does not affect the quality of the found solutions, i.e., the EDP values are as low as the default algorithm. Meanwhile, warm-start can converge **3.3x-7.3x** faster (we define time-to-converge as the time to reach 99.5% of performance improvement. In the figure we use the number of generation-to-converge, an equivalent index of time-to-converge.). We observe that Mnasnet [61] enjoys the least speedup. It is because Mnasnet is a result of neural architecture search, with irregular tensor shapes in each layer. Therefore scaling from previously-seen solutions will perform not as close to the optimized solutions as in regular networks such as Resnet [18], VGG [56], Mobilenet [53], which are manual designed. Nonetheless, warm-start for Mnasnet can still converge **3.3x** faster.

5.2. Sparsity-aware MSE

5.2.1. Motivation. In §4.5.2 we identified the need different mappings for different sparsity of workloads. While tackling weight sparsity is straightforward because weight sparsity is often fixed at model deploy time, tackling activation sparsity is challenging. Since the activation sparsity is not known a priori before runtime, and it differs per each input data, rather than asking MSE to search for the optimal mappings for all layers and all runtime dynamic sparsity levels, we ask MSE to search for “a sparsity-aware mapping” that is efficient across a range of sparsity levels. The only information the MSE relies on is what is the typical “range” of sparsity level for a given workload, e.g., 1.0 - 0.1 for a typical DNN workload.

It is not practical to search for an optimal mapping for each new input-activation. We want to seek out *if we can discover a mapping that can generalize across a range of sparsity levels to tackle the dynamic sparsity in activations?*

5.2.2. Proposed Sparsity-aware Search Mechanism. We propose sparsity-aware mapping search, which works as follows. When executing MSE, we don’t look at the actual density level of each activation (since it is dynamic). Instead, we assume and impose sparsity in the workload when executing MSE. We impose the activation to have a density from 1.0 to 0.1, which is the typical range of activation density in DNN [37, 45, 48, 74, 77, 78]. Next, when executing MSE, we score the mapping by the performance of this mapping on workload across the sweep of density levels (Fig. 8).

Scoring a Mapping. We score a mapping by the weighted sum of the performance. We use a heuristic that “the hardware performance (e.g., latency, energy) is with positive correlation to the density of the workload” to decide the

TABLE 4: Comparisons of sparsity-aware technique and static-density heuristic when tackling the activation sparsity. The static-density heuristic searches mapping for a fixed density level (1.0, 0.5, or 0.1). At search time, the sparsity-aware technique are enabled to see the performance of a mapping on a limited sets of density levels, which are randomly picked, e.g., 1.0, 0.8, 0.5, 0.2, and 0.1 in this experiments (marked as blue cells). We highlight the best-performing one in each row with green text. Sparsity-aware will find one fixed mapping solution. We test the found mapping with a range of density (1.0 - 0.05) and record their performance. Note that many of the density levels (in 1.0 - 0.05) are never seen by MSE at search time. The result indicates that sparsity-aware technique can find mapping with comparable performance to the static-density ones across a range of sparsity.

EDP (Energy uJ)				
Workload Density	Sparsity-aware	Static density 1.0	Static density 0.5	Static density 0.1
Resnet Conv_3, Accel-B				
1.0	2.40E+13	2.39E+13	2.41E+13	2.46E+13
0.9	1.75E+13	1.94E+13	1.76E+13	1.79E+13
0.8	1.23E+13	1.54E+13	1.24E+13	1.26E+13
0.7	8.26E+12	1.18E+13	8.30E+12	8.46E+12
0.6	5.21E+12	8.69E+12	5.24E+12	5.34E+12
0.5	3.02E+12	6.06E+12	3.02E+12	3.10E+12
0.4	1.55E+12	3.90E+12	1.56E+12	1.59E+12
0.3	6.59E+11	2.21E+12	6.63E+11	6.77E+11
0.2	1.98E+11	1.00E+12	1.99E+11	2.04E+11
0.1	4.78E+10	2.65E+11	4.81E+10	4.78E+10
0.05	1.28E+10	7.34E+10	1.29E+10	2.67E+10
Inception Conv_2, Accel-B				
1.0	7.77E+15	7.77E+15	7.93E+15	7.83E+15
0.9	5.67E+15	6.33E+15	5.79E+15	5.71E+15
0.8	3.99E+15	5.00E+15	4.08E+15	4.02E+15
0.7	2.67E+15	3.84E+15	2.74E+15	2.69E+15
0.6	1.69E+15	2.82E+15	1.73E+15	1.70E+15
0.5	9.78E+14	1.97E+15	9.78E+14	9.83E+14
0.4	5.02E+14	1.26E+15	5.21E+14	5.05E+14
0.3	2.13E+14	7.16E+14	2.23E+14	2.14E+14
0.2	6.39E+13	3.22E+14	8.64E+13	6.38E+13
0.1	1.55E+13	8.37E+13	4.49E+13	1.53E+13
0.05	4.12E+12	2.25E+13	2.53E+13	3.98E+12

weighting. We pick the weighting by the factor of $density^7$. For example, assuming we have two density levels, 0.5 and 1.0, with hardware performance $Perf_{0.5}$ and $Perf_{1.0}$, then the (weighted sum) score is: $\frac{Perf_{0.5}}{0.5} + \frac{Perf_{1.0}}{1.0}$.

5.2.3. Evaluation. We compare the “sparsity-aware” (§5.2.1) with “static-density” in Table 4. Both “sparsity-aware” and “static-density” are agnostic to the actual workload density. “Static-density 1.0” always assumes the workload is dense when searching. “Static-density 0.5” searches the mapping assuming the workload has 0.5 density, and “Static-density 0.1” assumes 0.1 density. “Sparsity-aware” searches the mapping assuming the workload density range from 1.0 - 0.1. Specifically, we use 5 density levels: 1.0, 0.8, 0.5, 0.2, and 0.1 (blue cells in the first column), which are picked by heuristics. That is, when evaluating the mapping in the

7. We pick the weighting linear to $density$, since we experiment only with activation sparsity (not weight) in our evaluation.

optimization loop, we scored the mapping by the performance of this mapping under workload density levels of 1.0, 0.8, 0.5, 0.2, and 0.1, and used the weighted sum of the performance as the final scores for the mapping. The scores are used to select which mappings proceed to the next iteration of the optimization loop.

We test the found mappings of the four strategies (columns) in Table 4 by workload with density from 1.0 to 0.05. The performance of each is recorded in the corresponding rows. We make two observations: 1) The “sparsity-aware” can reach comparable performance to the “static-density” ones at the density levels, for which the “static-densities” are specifically optimized. For example, “static-density 1.0” found a mapping with EDP 2.39E+13 (cycles uJ) at density level 1.0. The mapping found by “sparsity-aware” can perform at a comparable EDP of 2.40E+13 (cycles uJ). 2) Aware of a range of sparsity (1.0 - 0.1), “sparsity-aware” can successfully find a mapping that can generalize across a range of sparsity. A fixed mapping found by “sparsity-aware” can achieve (in geomean) **99.7%** of performance to the performance of each of the mappings specifically searched for different density levels.

6. Related works

Map Space Exploration. Many mappers (search algorithms) with different algorithmic techniques are proposed to tackle the MSE problem. Timeloop-mapper [44], Simba [54], dmazeRunner [12], Interstellar [75], and others [13, 14, 41, 55, 57–60, 63, 66, 67, 70, 76] use random sampling on a raw or pruned search space. Gamma [28], Autotvm [7], and others [30, 60, 64] use genetic algorithms. Tiramisu [3] and Tensor Comprehensions [64] use constrained optimization. HASCO [73] and Reagen et. al [50] uses Bayesian optimization, RELEASE [2], ConfuciuX [27], and FlexTensor [79] uses reinforcement learning. Mind Mappings [19] uses a neural network-based surrogate model to replace the cost model and directly uses backpropagation to learn a solution that maximizes the objective. There are also other techniques such as mixed-integer programming in CoSA [23], MCMC search in FlexFlow [25], and others [3, 15, 49, 64]. While there have been plenty of mappers proposed, a deeper analysis of how the MSE works and how different mapping axes contribute to the performance is often lacking, which this work performs.

7. Conclusion

MSE for NPUs is a computationally expensive problem with active ongoing research. There is, however, no work, to the best of our knowledge, that has focused on understanding how different state-of-the-art mappers navigate the map-space across different axes. This work performs a deep-divide analysis on MSE using heuristic and learning-based mappers and identifies their strengths and weaknesses. We also propose two new techniques - warm-start and sparsity-aware - to enable scalability to emerging large, irregular and sparse DNNs. We hope that by our analysis, we can make MSE more approachable and understandable to a broader

community, and propel the invention of advanced mapping search techniques.

Acknowledgments

We thank Yannan Wu for the advice and support on Sparseloop setup. This work was supported in-part by NSF Award #1909900.

References

- [1] “Nvidia deep learning accelerator,” <http://nvidia.org>, 2017.
- [2] B. H. Ahn, P. Pilligundla, and H. Esmailzadeh, “Reinforcement learning and adaptive sampling for optimized dnn compilation,” *arXiv preprint arXiv:1905.12799*, 2019.
- [3] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, “Tiramisu: A polyhedral compiler for expressing fast and portable code,” in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 193–205.
- [4] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, “Once-for-all: Train one network and specialize it for efficient deployment,” *arXiv preprint arXiv:1908.09791*, 2019.
- [5] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” *arXiv preprint arXiv:1812.00332*, 2018.
- [6] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, “Marvel: A data-centric compiler for dnn operators on spatial accelerators,” *arXiv preprint arXiv:2002.07752*, 2020.
- [7] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Learning to optimize tensor programs,” in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.
- [8] Y.-H. Chen *et al.*, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *JSSC*, vol. 52, no. 1, pp. 127–138, 2016.
- [9] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, “Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.
- [10] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.
- [11] R. Child, S. Gray, A. Radford, and I. Sutskever, “Generating long sequences with sparse transformers,” *arXiv preprint arXiv:1904.10509*, 2019.
- [12] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, “Dmazerunner: Executing perfectly nested loops on dataflow accelerators,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–27, 2019.
- [13] M. Gao *et al.*, “Tangram: Optimized coarse-grained dataflow for scalable nn accelerators,” in *ASPLOS*, 2019, pp. 807–820.
- [14] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.
- [15] T. Gresser, H. Zheng, R. Aloor, A. Simbürger, A. Gröbinger, and L.-N. Pouchet, “Polly-polyhedral optimization in llvm,” in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

- [16] F.-M. Guo, S. Liu, F. S. Mungall, X. Lin, and Y. Wang, "Reweighted proximal pruning for large-scale language representation," *arXiv preprint arXiv:1909.12486*, 2019.
- [17] N. Hansen, "The cma evolution strategy: a comparing review," in *Towards a new evolutionary computation*. Springer, 2006, pp. 75–102.
- [18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [19] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 943–958.
- [20] M. Hellwig *et al.*, "Evolution under strong noise: A self-adaptive evolution strategy can reach the lower performance bound-the pcmsas," in *International Conference on PPSN3*. Springer, 2016, pp. 26–36.
- [21] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.
- [22] 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 *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022, pp. 254–266.
- [23] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzyniak, T. Norell, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 554–566.
- [24] G. Jeong, G. Kestor, P. Chatarasi, A. Parashar, P.-A. Tsai, S. Rajamanickam, R. Gioiosa, and T. Krishna, "Union: A unified hw-sw co-design ecosystem in mlir for evaluating tensor operations on spatial accelerators," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 30–44.
- [25] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.
- [26] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, 2019, pp. 600–614.
- [27] S.-C. Kao, G. Jeong, and T. Krishna, "Confucius: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 622–636.
- [28] S.-C. Kao and T. Krishna, "Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm," in *ICCAD*, 2020.
- [29] —, "E3: A hw/sw co-design neuroevolution platform for autonomous learning in edge device," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 288–298.
- [30] —, "Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores," *IEEE International Symposium on High-Performance Computer Architecture (HPCA-28)*, 2022.
- [31] S.-C. Kao, M. Pellauer, A. Parashar, and T. Krishna, "Digamma: Domain-aware genetic algorithm for hw-mapping co-optimization for dnn accelerators," *Design, Automation and Test in Europe Conference (DATE)*, 2022.
- [32] S.-C. Kao, S. Subramanian, G. Agrawal, and T. Krishna, "An optimized dataflow for mitigating attention performance bottlenecks," *arXiv preprint arXiv:2107.06419*, 2021.
- [33] J. Kennedy *et al.*, "Particle swarm optimization," in *ICNN*, vol. 4. IEEE, 1995, pp. 1942–1948.
- [34] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 754–768.
- [35] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.
- [36] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.
- [37] C.-E. Lee, Y. S. Shao, J.-F. Zhang, A. Parashar, J. Emer, S. W. Keckler, and Z. Zhang, "Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in *SysML Conference*, vol. 120, 2018.
- [38] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.
- [39] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 19–34.
- [40] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.
- [41] W. Lu *et al.*, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *HPCA*. IEEE, 2017, pp. 553–564.
- [42] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021.
- [43] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.
- [44] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 304–315.
- [45] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.
- [46] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 137–151.
- [47] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *arXiv preprint arXiv:1802.03268*, 2018.
- [48] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training."
- [49] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

- [50] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2017, pp. 1–6.
- [51] H. Sajjad, F. Dalvi, N. Durrani, and P. Nakov, "Poor man's bert: Smaller and faster transformer models," *arXiv preprint arXiv:2004.03844*, 2020.
- [52] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.
- [53] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.
- [54] Y. S. Shao *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *MICRO*, 2019, pp. 14–27.
- [55] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.
- [56] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
- [57] L. Song *et al.*, "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *HPCA*. IEEE, 2019, pp. 56–68.
- [58] M. Song *et al.*, "Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning," in *HPCA*. IEEE, 2018, pp. 66–77.
- [59] A. Stoutchinin *et al.*, "Optimally scheduling cnn convolutions for efficient memory access," *arXiv preprint arXiv:1902.01492*, 2019.
- [60] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA'16*, 2016, pp. 16–25.
- [61] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.
- [62] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.
- [63] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.
- [64] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.
- [65] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
- [66] S. Venkataramani *et al.*, "Scaleddeep: A scalable compute architecture for learning and evaluating deep networks," in *MICRO*, 2017, pp. 13–26.
- [67] —, "Deeptools: Compiler and execution runtime extensions for rapid ai accelerator," *IEEE Micro*, vol. 39, no. 5, pp. 102–111, 2019.
- [68] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," *arXiv preprint arXiv:2012.09852*, 2020.
- [69] Z. Wang, J. Wohlwend, and T. Lei, "Structured pruning of large language models," *arXiv preprint arXiv:1910.04732*, 2019.
- [70] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *DAC*, 2017, pp. 1–6.
- [71] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 232–234.
- [72] —, "Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling," in *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2022.
- [73] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang, "Hasco: Towards agile hardware and software co-design for tensor computation," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1055–1068.
- [74] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 236–249.
- [75] X. Yang *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *ASPLOS*, 2020, pp. 369–383.
- [76] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.
- [77] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "Snap: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, 2020.
- [78] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
- [79] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 859–873.
- [80] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.