Demystifying Map Space Exploration for NPUs

Sheng-Chun Kao* Angshuman Parashar† Po-An Tsai† and Tushar Krishna*

*Georgia Institute of Technology
†Nvidia

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 mapper on an 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 ~ $O(10^{24})$ [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.

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

Fig. 1: The overview of DNN Workload, Accelerator, and a (NVDLA-like [1]) Mapping.
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 to represent each tiling level (e.g., NVDA-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.

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.

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] used (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.

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 $2^{7×2}$.

Map-Space. The Cartesian product of these sub-spaces leads to the size of the entire map space, which is at the level of $O(10^{42})$ 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.

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

<table>
<thead>
<tr>
<th>Workload</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resnet Conv_3</td>
<td>(16,128,128,28,28,3,3)</td>
</tr>
<tr>
<td>Resnet Conv_4</td>
<td>(16,256,256,14,14,3,3)</td>
</tr>
<tr>
<td>Inception Conv_2</td>
<td>(16,192,192,27,27,5,5)</td>
</tr>
<tr>
<td>Workload</td>
<td>(8,B,M,K,N)</td>
</tr>
<tr>
<td>Bert-Large KQV</td>
<td>(16,1024,1024,512)</td>
</tr>
<tr>
<td>Bert-Large Attn</td>
<td>(16,612,1024,512)</td>
</tr>
<tr>
<td>Bert-Large FF</td>
<td>(16,4096,1024,512)</td>
</tr>
<tr>
<td>Accel A</td>
<td>512 KB shared buffer,</td>
</tr>
<tr>
<td></td>
<td>64 KB private buffer per PE, 256 PEs, 1</td>
</tr>
<tr>
<td></td>
<td>ALUs per PE</td>
</tr>
<tr>
<td>Accel B</td>
<td>64 KB shared buffer, 256 B private buffer</td>
</tr>
<tr>
<td></td>
<td>per PE, 256 PEs, 4 ALUs per PE</td>
</tr>
</tbody>
</table>

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.

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.
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 a collected training dataset. The training dataset includes
collected workloads and collected accelerator configurations. While surrogate model can generalize the workload encoding with the lower-performing region and gradient-updates to the workload and accelerator configuration in Fig. 3(a). Fig. 4(a) one of the high-performance regions, as shown in Fig. 4(b). feedback-based algorithms [17, 20, 21, 33]). Gamma reached region but can explore a wider region faster because of local optimum. Gamma also starts with a lower-performing higher-performing regions at the right. However, it sits at the most of the design points sit here. Mind Mappings starts only samples around the lower-performing region because limited 5,000 sampling budget, The Random-Pruned method the points different algorithms actually sampled. Given the points (green circle) scatter across the space. Fig. 4(b) shows the center while some small clusters of the high-performing performance points. There is a large low-performing region at shows the entire map space while dark red represent higher-performance regions. 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 disabling crossover
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).

(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 group 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 to relax the complexity, which becomes 7!=5,040 choices.

**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
In Fig. 9, we show the performance of the initialized mapping of warm-start (yellow bar), warm-start by previous layers, 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 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. 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 convergence curve of the first layer and a later layer of 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 weighting. We pick the weighting by the factor of density. For example, assuming we have two density levels, 0.5 and 1.0, with hardware performance Perf_M and Perf_1.0, then the (weighted sum) score is: \( \frac{0.5 \times Perf_{0.5} + 1.0 \times Perf_{1.0}}{1.0} \).

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 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.

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 weighting. We pick the weighting by the factor of density. 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: \( \frac{0.5 \times Perf_{0.5} + 1.0 \times Perf_{1.0}}{1.0} \).

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
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-dive 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 Sparselooop setup. This work was supported in-part by NSF Award #1909900.

References
