ABSTRACT

DNN layers are multi-dimensional loops that can be ordered, tiled, and scheduled in myriad ways across space and time on DNN accelerators. Each of these choices is called a mapping. It has been shown that the mapping plays an extremely crucial role in overall performance and efficiency, as it directly determines the amount of reuse that the accelerator can leverage from the DNN. Moreover, instead of using a fixed mapping for every DNN layer, research has revealed the benefit of optimizing per-layer mappings. However, determining the right mapping, given an accelerator and layer, is still an open question. The immense space of mappings (or map-space) makes brute-forced exhaustive search methods unapproachable. In this paper, we propose a domain-specific genetic algorithm-based method, GAMMA, which is specially designed for this HW-mapping problem. In contrast to prior works that either target simple rigid accelerators with a limited map-space or choose from a restricted set of mappings, we construct an extremely flexible map-space and show that GAMMA can explore the space and determine an optimized mapping with high sample efficiency. We quantitatively compare GAMMA with many popular optimization methods and observe GAMMA consistently finds better solutions.

CSCS CONCEPTS

- Hardware → Hardware accelerators.

KEYWORDS

Genetic Algorithm, ML accelerator, Reconfigurable device

ACM Reference Format:


1 INTRODUCTION

Deep neural networks (DNNs) are being deployed into many real-time applications such as autonomous driving, mobile VR/AR, and…

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICCAD ’20, November 2–5, 2020, Virtual Event, USA.
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8026-3/20/11 . . $15.00
https://doi.org/10.1145/3400302.3415639
HW-mapping search space (exceeding $O(10^{36})$) even for a single layer of a DNN, as shown later in Section 2.6) makes the problem highly challenging. To cope with this challenge, most prior works restrict the search space. For e.g., coarse-grained strided exhaustive search [35, 47, 52, 54, 56], random search [35], fixed parallelism [32, 34, 52, 54, 56, 68], or limited search for tile sizes for one or more fixed dataflows [33, 66]. Alternately, ML-based search techniques have also been leveraged for guided search to increase sampling efficiency [4, 8, 41, 61]; however, they need to restrict some aspects of the mapping space (e.g., fixing the parallelism levels) to adapt to the ML algorithms. Such restrictions of the mapping space can lead to local optimal mappings which are significantly sub-optimal, as recent works have highlighted [35, 46].

To efficiently deal with the massive search space of HW-mappings, we propose GAMMA (Genetic Algorithm-based Mapper for ML Accelerators). Unlike prior works, GAMMA performs a complete search, considering all three aspects of HW-mapping (tiling strategy, computation order, and parallelization strategy). Furthermore, GAMMA can explore up to three levels of parallelism within the mapping, as shown in Fig. 1(b), unlike prior works that target one to two levels. Thus GAMMA can work across both single-accelerator [11, 25] and multi-accelerator [46] systems. The key novelty in GAMMA is (i) a specialized genetic encoding of all three aspects of HW mapping; (ii) specialized mutation and crossover operators to evolve new mappings, and (iii) new genetic operators to model the behavior of adding and removing levels of parallelism. We also develop a closed-loop workflow by integrating GAMMA with a popular analytical cost-model for DNN mappings called MAESTRO [2] to fully automate the mapping search problem.

We examine GAMMA on multiple popular DNN models, VGG16 [49], MobileNet-V2 [44], ResNet-18 [21], ResNet-50 [21], and MnasNet [60]. We make quantitative comparisons with eight popular optimization methods[13, 20, 22, 23, 35, 42]. We observe that GAMMA can find HW-mapping with the HW performance (latency/energy) consistently better than other methods. Further, we show that GAMMA can be run over multiple stages, enabling it to leverage throughput slack from non-bottlenecked DNN layers and further reduce total system energy by 58% (for ResNet-18) and 78% (for VGG16) and power by 95% (for ResNet-18) and 99% (for VGG16). The contributions of this paper are as follows:

- **Comprehensive Map Space.** GAMMA constructs and searches through a comprehensive map-space comprising of computation order, tile-sizes, parallelization strategy, and up to three parallelization levels, enabling it to target a wide variety of fixed and flexible single and multi-accelerator systems.

- **Generic Encoding scheme.** The proposed encoding scheme transforms the HW-mapping problem to an optimization problem, which enables the user to directly use off-the-shelf optimization algorithms for mapping. These form our baselines.

- **New Genetic Algorithm operators.** GAMMA introduces three new GA operators, enabling a domain-specific flexible search space unlike most off-the-shelf optimization algorithms.

- **Autonomous Workflow.** We automate GAMMA as a blackbox optimizer for the HW-mapping problem. This reduces the learning curve and saves manual-tuning effort for ML practitioners exploring the HW-mapping space. GAMMA encapsulates an end-to-end workflow, which generates outputs compatible with an open-source cost model [2]. We will open-source the GAMMA infrastructure after the paper gets published.

The paper is organized as follows. Section 2 provides relevant background on DNN accelerators and optimization methods; Section 4 describes GAMMA; Section 5 presents comprehensive evaluations; Section 6 discusses related works and Section 7 concludes.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Layer types in DNNs

There are myriads of DNN models and most of them are built using different combinations of some common layers. Convolutional layers (2D/depth-wise/point-wise) dominate in DNNs like ResNet-50 [21], MobileNet-V2 [44], and InceptionNet [58] targeting image processing tasks. Fully connected layers or MLPs are often used as the last layer in many DNNs and as hidden layers of RNNs. Different layer types expose different amounts of data reuse opportunities, which can be exploited by DNN accelerators. In this work, we consider both CNNs and MLPs that dominate modern DNNs.

### 2.2 DNN Accelerator Architectures

#### 2.2.1 HW resources. Spatial DNN accelerators comprise of a 2D array of Processing Elements (PEs), as shown in Fig. 1(c). Each PE has a MAC to compute partial sums, and local scratchpad (SL) buffers to store weights, activations, and partial sums. The accelerators also house a global scratchpad (SG), shared among PEs, to prefetch activations and weights from DRAM for the next tile of computation that will be mapped over the PEs and SL buffers.

#### 2.2.2 Dataflow. In addition to the HW resources, each accelerator needs to select a dataflow strategy to stage data movement in order to leverage data reuse. Dataflow comprises of computation order and parallelism strategy, which we describe in Section 2.3. It directly affects the amount of data reuse the accelerator is able to leverage. Dataflows can be fixed at design-time [1, 11, 15] or configured at compile-time [30, 33]. In this work, we assume an accelerator with configurable dataflows.

### 2.3 Mapping DNNs over Accelerators

Mapping refers to the dataflow strategy (i.e., computation order and parallelism) coupled with the tiling strategy [31, 35]. We describe the three components of a DNN mapping next.

#### Computation order. 2D convolution, shown in Fig. 1(a), is a six loop-nest (or seven loops when considering batching). We can swap the order of these loops randomly, incurring 6! choices. Different orderings lead to different temporal reuse (aka "stationary" [10]) or spatial reuse (multi-casting) opportunities for the operands.

#### Parallelism strategy. The parallelism strategy comprises of (i) the number of levels of parallelism, and (ii) the specific loops to parallelize or spatially unroll [67] at each level). Different dimensions of parallelism at each level incur different data movement patterns, multi-casting behavior, and reuse opportunities [31]. A simple example is the NVDLA [1] architecture that employs a fixed 2-level mapping along the K and C dimensions for convolutions.

The number of parallelism levels depends on the number of independent spatial dimensions within the accelerator substrate. In this work, we assume up to three levels of parallelism (e.g., rows x cols x multiple arrays) as shown in Fig. 1(b). This sets the possible
choices to $6 \times 5 \times 4 = 120$. A 3-level mapping can work on a flexible accelerator fabric [30] or multiple 2D accelerators (Fig. 1(c)).

**Tiling strategy.** We refer to tiling strategy as designing the tiling size of each dimension (e.g., input(C)/output(K) channel dimensions, Y/X dimensions of activations, R/S dimensions of weights in Fig. 1(a)). The tile size of each dimension could range from 1 to the size of the dimension. For e.g., the second layer of VGG16 [49], ($K=64, C=64, Y=224, X=224, R=3, S=3$), forms a tile size search space of $O(10^9) = 64^2 \times 224^2 \times 3^5$.

The tile sizes determine the number of operands of each tensor (input/output) that need to be present within the accelerator buffers at each time-step. Tile sizes depend on both the dataflow strategy and the size of the buffers. Tiles of data move between DRAM and the SG buffers, and between the SG and SL buffers. (As discussed above, some of these data items may remain stationary in the buffers for longer, while some stream in and out, depending on the computation order). The rate at which different tiles move determines the off-chip and on-chip bandwidth requirement; bandwidth less than this leads to stalls.

### 2.4 Target Accelerator Systems

Many DNN accelerators come with a fixed dataflow strategy baked into silicon [1, 11, 15, 25]. The job of a mapper is to simply find tile sizes for each layer to fit within the buffers. However, there is no perfect dataflow that is supreme for all layers in a DNN model [31, 35]. This led to a suite of flexible accelerators [30, 33] that allow dataflow configurability [31, 35] at compile-time via flexible buffering and connectivity to make the accelerator future-proof to emerging DNNs. This work considers such accelerators and hence assumes that the mapping (computation order, parallelizing dimensions and tile sizes) are configured at compile-time. The only constraint for the mapper from hardware is the maximum number of parallelism levels (which depends on the accelerator array flexibility) and maximum tile-sizes (limited by the buffer sizes). It is certainly possible to restrict further aspects of the mapping, if the hardware desires, within our framework. Alternately, our proposed framework can also be used at design-time to determine the optimal dataflow for accelerators built for specific DNNs types. We present our target accelerators in Section 4 and Table 3.

### 2.5 MAESTRO: A Cost Model for evaluating the cost of DNN Mappings

Frameworks like MAESTRO [2] and Timeloop [35] use detailed analytical modeling to evaluate different mapping strategies of a DNN on the accelerators. We leverage MAESTRO [2] as our underlying cost model because of its ability to support the target detailed mapping space. It supports most of the common DNN layers such as CONV, depth-wise CONV, and Fully connected. Given a DNN layer, a HW resource configuration (PE, SL size, SG size, NoC latency and bandwidth), and a mapping strategy, MAESTRO estimates the statistics such as latency, energy, runtime, power, and area.

**Impact of Mapping on Performance and Energy.** In Fig. 2(a) and (b), we randomly sampled 10K possible HW mappings and use the cost model to estimate its corresponding HW performance. Each datapoint reflects a valid HW mapping design for the same DNN layer (the second layer of VGG16). From Fig. 2(c), we can observe several order of difference on HW performance when the mapping varies. This validates similar studies [35] and shows how critical the HW-mapping is to the performance of a DNN accelerator.

#### 2.6 Mapping Search-Space aka Map Space

The mapping of a 1-level mapper in the example of the second layer of VGG16 is $O(10^{12}) = O(10^9 \times 6! \times 6)$. A N-level mapper increases the mapping space by the power of N, leading to $O(10^{12N})$, which is $O(10^{36})$ when considering three level of parallelism. This design space is hard to enumerate even with coarse-grained stridden enumeration. Therefore, a domain-specific specialized optimization algorithm is needed to search the design space with sample efficiency.

### 2.7 Baseline Search Methods

Many search/optimization methods exist today for architects to perform Design-Space Exploration (DSE) and form our baselines. We leverage eight optimization methods, including Random Search, Genetic Algorithm (GA) [23], Differential Evolution (DE) [38], (1 + λ)-ES [42], Covariance matrix adaptation evolution strategy (CMA-ES) [20], Test-based Population-Size Adaptation (TBPSA) [22], Particle Swarm Optimisation (PSO) [26], Passive Portfolio (pPortfolio)[13]. We summarize them in Table 4.

### 3 GAMMA ALGORITHM AND WORKFLOW

#### 3.1 Challenges with Baseline Methods

The baseline methods in Section 2.7 can all be used for HW-mapping search. However many algorithms (e.g., CMA-ES, PSO, DE, and also standard GA) work in rigid search space, i.e., the number of parameters in a design point is pre-defined, which restricts the levels of parallelism (Section 2.3) to a pre-defined number and shrinks the potential search space by log scale. Our goal is to parameterize the level of parallelism as well. We need a framework that accepts input with flexible lengths. There are many possible ways to realize such a flexible framework, such as adding an extra auto-encoder [27] or using an sequence-to-sequence structure [57] at the input layer. However, they require another level of optimization, training, or approximation, which brings in relatively large overhead comparing to the optimization algorithm (DE, ES, standard GA) itself.

---

**Table 1: Terminology in Genetic Algorithm (GA).**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene</td>
<td>The encoded value of one of the dimensions of a design point.</td>
</tr>
<tr>
<td>Genome (Individual)</td>
<td>A complete series of genes representing a design point.</td>
</tr>
<tr>
<td>Elite</td>
<td>A set of genomes that has the highest evaluated fitness.</td>
</tr>
<tr>
<td>Population (Generation)</td>
<td>An entire set of genomes forms a population (one generation). The populations evolve with time by mutation/crossover and selection of the well-performing genomes to the next generation.</td>
</tr>
<tr>
<td>Crossover</td>
<td>Blend two parents’ genes to reproduce children genomes.</td>
</tr>
<tr>
<td>Mutation</td>
<td>Randomly perturb a parent’s genes to reproduce children genomes.</td>
</tr>
</tbody>
</table>
Figure 3: The GAMMA encoding example of (a) 1-level mapper and (b) 2-level mapper.

Figure 4: (a) GAMMA’s description of a 2-level mapper and (b) its decoded description for cost model (MAESTRO) of a NVDLA-like DNN model/DNN layers.

Figure 5: (a) The structure and algorithm flow of GAMMA, and (b) The summary of evolution in GAMMA.
Initialization. Assuming population size \( P \), we randomly initialize \( P \) number of the 1-level mappers. The only restriction is each tile size is smaller than the corresponding layer dimension.

Evolution: Crossover. Crossover is to take advantage of the genes in some well-performing genomes, which forms a parents subset. We randomly pick two genomes from the parents subset. We blend their genes by interchanging the value of the tile size.

Evolution: Mutation - Parallel Dim. With a certain probability, which is set by the mutation rate of the algorithm, we mutate the parallelism dimension by randomly sampling one of the 6 dimensions of the tensor and setting it as a new parallelism dimension.

Evolution: Mutation - Tile Size. With a certain probability, we randomly pick paired genes and assign a new random tile size for them. If the tile-size in the mapping does not fit within the SL buffer of the PE for that operand, it is given a large penalty during its evaluation, as we discuss later in Section 3.6.3.

Evolution: Reorder. Reorder is another format of mutation. We pick two paired genes and swap their position in the genome, which reflect the reordering of the mapping.

Evolution: Growth. With a certain probability, we grow the genome by appending a randomly initialized 1-level genome to the current genome, as shown in Fig. 3(b). The original L1 mapper will be promoted to L2-mapper, and the newborn genome is noted as the new L1-mapper.

Evolution: Aging. The natural phenomenon of a person’s DNA keep shortening in the lifespan is known as DNA aging. With a certain probability, we will “age” the genome by cutting out the tail of genome, an L1-mapper, which moves genome from (b) back to (a) in Fig. 3.

Evaluate and Selection. After evolution, we evaluate the populations by interacting with the evaluation environment (Env), which we will describe in Section 3.6.3. Env will feedback the fitness of each individual. We select the population that is eligible to enter the next generation by the ranking of their fitness.

3.6 Flow for Automated Mapping Search

3.6.1 Constraint. Our target is to find the HW mapping of a DNN layer that fits within limited HW resources - PEs and buffers. Different mapping lead to drastically different requirement of HW resources, especially the buffer sizes, as shown in Fig. 2(c). Note that a mapping implicitly runs over multiple time iterations if the number of computations in the dimension to parallelize exceeds the number of available PEs; however, the local (SL) buffer and global (SG) buffer sizes to run each iteration (which comes from the tile sizes) is a hard constraint when searching through the map-space.

3.6.2 Objective. The target is to minimize the objective. The objective could be any HW performance index that the user is interested in such as latency, power, energy, area, energy-delay-product (EDP), or other combinations of them. Minimizing the objective is not a trivial task since there is no straight-forward solution even for a common tensor shape of a DNN layer. Minimizing the latency as an example, the most efficient choice of parallelizing dimension involves the shape of tensor, available PEs to parallelize, available SL/SG buffers to house the fetched data, and the tile size of each dimension. All the decisions (or genes) correlate and jointly decide the latency. Some common heuristics of parallelizing across activations dimensions at the early layers and across channel dimensions at the late layers in CNN becomes challenging when involving HW resource constraint and the multi-level parallelism flexibility of mapping strategy.

3.6.3 Interactive Environment (Env). Structure: The Env is initialized with the target DNN layer, the accelerator constraint, and the optimization objective (latency/energy/power). Env contains a HW performance cost model, where we leverage MAESTRO [2] for its ability to model and evaluate arbitrary spatial accelerators and mappings. When interacting with GAMMA, Env takes in an entire generation of populations, decodes them into the input format of the cost model as describe in Section 3.4, and feeds into the cost model to gather the statistics of their HW performance. Finally, using the fitness function, which we discuss next, Env extracts fitness scores and returns them to GAMMA.

Fitness Function: We extract the corresponding reward value (= -Perf. index) from the statistics according to the set objective, and substitute it into the fitness function. We give the individual a large penalty - a negative infinite - when the constraint is not met. That is, the evolved mapping require more HW resources than the accelerators’ constraint, which is then not suitable for the targeting accelerators. The fitness function is summarized as following.

\[
\text{Fitness} = \begin{cases} 
\text{reward, if constraint met} \\
\text{Infinitive, others}
\end{cases}
\]

4 METHODOLOGY

4.1 Models and Platforms

DNN Models. In our experiment, we consider five CNN models with different complexity: VGG16 [49], MobileNet-V2 [44], ResNet-50 [21], ResNet-18 [21], MnasNet [60].

HW resources of Platforms. We consider two platforms with different number of HW resource: cloud platform (which resembles the HW resources in Eyeriss chip[11]), as shown in Table 2. Target Systems. We consider three kinds of accelerator systems as shown in Table 3.

4.2 Target Search Methods and Parameters.

We compare three sets of methods, described below. We set the maximum sampling points as 10K for all methods and compare the HW performance of their searched solutions.

Baseline Optimization Methods. We compare with a suite of optimization methods whose implementations are adopted from Nevergrad [40]. The methods, and their experimental parameter settings are summarized in Table 4.

Fixed Dataflows from prior accelerators. We also compare with some widely recognized HW-mappings inspired by dataflows
We record the best solution (lowest latency) of each algorithm after properly formulating it into an optimization problem, which can significantly relieve the domain expert’s effort on the back-and-forth tuning process. The challenge is the occasional failing of some methods. Moreover, GAMMA can consistently find valid and better solutions than others. Comparing with others, GAMMA finds solutions costing $224\times$ to $440\times$ less latency in Edge platform and $153\times$ to $(1.3E+7)\times$ less latency in Cloud platform.

### 5.2 S2: Flexible 2D Accel.

**5.2.1 Objective: Latency.** Fig. 6(c)-(d) compares latency for accelerators with flexible aspect ratios (i.e., the accelerator can support both 1-level and 2-levels of parallelism). One interesting observation is that the fixed ShiDianNao-like dataflow and NVDLA-like dataflow shows better performance than baseline optimization methods at early and late layers respectively. This is because ShiDianNao-like dataflow parallelizes along X, Y dimensions and early layers of ResNet-18 have high X-Y values; similarly NVDLA-like parallelizes along C-K and late layers have high C-K values. For the baseline methods, since the number of parallelism dimensions is fixed, we search for the best 1-level and 2-level solution for 5K points each, and pick the better one. GAMMA searches across both 1-level and 2-level via the growth and aging operators described earlier. GAMMA finds valid and better solutions than others. Compared with other techniques, GAMMA finds solutions with $209\times$ to $1.035\times$ less latency in Edge and $337\times$ to $(7.1E+5)\times$ less latency in Cloud platform.

**5.2.2 Objective: Energy.** Energy consumption depends on the number of active computations, memory accesses, and SL/SG buffer usages. In the Edge platform in Fig. 8(a), fixed dataflow show no advantage, and most optimization methods can find better solutions than the fixed dataflows. In the Cloud platform in Fig. 8(b), optimization methods show competitive or better performance than Eyeriss-like and ShiDianNao-like dataflow. However, NVDLA-like dataflow shows high energy efficiency, since the K, C dimensions expand in ResNet-18, which gives more advantage to the dataflow that is skilled at layer with large K, C dimensions (NVDLA-like). They finish the computation with shorter time and less memory access, and hence cost less energy. However, these advantages do not show up when NVDLA-like dataflow is in the tight constraint (Edge platform), which has less SL/SG buffer and limited the parallelism opportunity. Across all fixed dataflow and optimization methods, GAMMA finds solutions costing $11\times$ to $36\times$ less energy in Edge platform and $2\times$ to $42\times$ less energy in Cloud platform.

### 5.3 S3: Scale-out Flexible 2D Accel.

**5.3.1 The growth of search space.** Increasing the level of parallelism will exponentially increase the search space, which makes the performance of the optimization methods more critical to the found solutions. As shown in Fig. 6(e)-(f), the number of cases that methods fail to find solution becomes significant. However, GAMMA can consistently find valid and better solutions, costing $241\times$ to $644\times$ less latency in Edge platform and $657\times$ to $(1.2E+5)\times$ less latency in Cloud platform. Fig. 6(g)-(h) shows the end-to-end latency of GAMMA in different accelerator systems. We can find GAMMA performs the best in S3, where the design space is several order larger than S1 and S2 but with more flexibility. It shows that GAMMA can explore the design space with sample efficiency and takes advantage of the flexibility of the mapping space.

**5.3.2 Deep-dive into found solution.** Fig. 7 shows the HW-mapping solutions found by GAMMA on ResNet-18. At the early layer (Y, X dominant, Y=224, X=224), GAMMA found a mapper that parallelizes

### Table 4: Baseline optimization methods.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>Random Search. We randomly sample design points and keep the best solution.</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm. We evolve the design point into a series of genes. We evolve the genes by mutation and crossover. We vary mutation rate: 0.1, crossover rate: 0.1, in the experiment.</td>
</tr>
<tr>
<td>DE</td>
<td>Differential Evolution. In DE, we mutate by the difference vector (DQ). When mutation, we sample a parent and extract its difference on each dimension with another parent to form a local DW and with the best parent to form a global DW. Next, we sample another parent and mutate it by adding the weighted sum of found local DW and global DW to it. We vary weighting for local DW: 0.8, weighting for global DW: 0.8, in the experiment.</td>
</tr>
<tr>
<td>1ES</td>
<td>(1+1)-Evolution Strategy. For each parent we mutate it to reproduce N numbers of mutated children. The parent and children compete with each other by their fitness values and the best one will go to the next generation. We vary (1+1)-ES in the experiment, where one parent generate one child.</td>
</tr>
<tr>
<td>CMA-ES</td>
<td>Covariance Matrix Adaptation-ES. We use covariance matrix of the elite and the entire population to estimate the distance to the optimum. We adapt the step size based on covariances. We use 1/2 of the best performing individuals as elite group in the experiment.</td>
</tr>
<tr>
<td>TPB</td>
<td>Test-based Population-Size Adaptation. We estimate the trend of the population’s fitness value. When the fitness values converge, the algorithm will adapt to have a smaller population size. We set the initial population size as 50 and let it evolve in the experiment.</td>
</tr>
<tr>
<td>PSO</td>
<td>Particle Swarm Optimization. We track global best and parent best, which represent the global and local information respectively. We update the parameters by the weighted sum of them. We use weighting for global best: 0.8, weighting for parent best: 0.8, with momentum: 0.8, in the experiment.</td>
</tr>
<tr>
<td>pPortfolio</td>
<td>Parent-Particle Portfolio. We maximize diversity by spreading the risk broadly to multiple instances (optimizers). We launch K numbers of optimization methods, each experimenting 1K samples, and take the best performing solution provided by one of them. We use the passive portfolio of CMA-ES and DE in the experiment.</td>
</tr>
</tbody>
</table>

Table 4: Baseline optimization methods.
Figure 6: The performance of found solutions across a suite of optimization methods on different target systems (S1, S2, S3) and different platform constraints (Edge, Cloud) for ResNet-18.

Table 5: End-to-end performance and energy on S3 for a suite of DNN models using fixed mappings versus GAMMA. Bold means lowest values.

Table 6: Summary of performance benchmarks on ResNet-18.

Figure 7: GAMMA's found mapping of early, medium, and late layer

Figure 8: The energy consumption of S2 on ResNet-18.

Figure 9: End-to-end latency improvement over generations with GAMMA for S3 system and edge platform constraint.

5.3.3 Other DNN models and end-to-end performance. Table 5 shows the performance of GAMMA comparing to fixed dataflow on other widely-used DNN models. Here, for the interest of space, we only list the end-to-end performance, which is the sum of latency/energy of all the layers. Table 5 shows no fixed dataflow is good across all DNNs and platforms. For example, when considering latency, ShiDianNao-like performs the best on Edge platform, and NVDLA-like performs the best on Cloud platform. In contrast, the energy numbers follow NVDLA-like < Eyeriss-like < ShiDianNao-like; NVDLA-like gets advantage over the other two for energy via reuse across K and C dimensions (which dominate in most CNN-based models). Among all experiments in Table 5, GAMMA almost always provides the lowest latency and energy. Across models and platforms, GAMMA finds solutions costing 5x to (1.2E+5)x less latency and 2x to (1.6E+4)x less energy. Fig. 9 tracks how GAMMA converges to its solution across generations; this shows its sample efficiency via rapid improvement over generations.

1The found solutions also show that by relaxing some tile size heuristics such as deciding tile size by the integral multiple of PEs array sizes could help reach better solutions.
Table 6: Two stage optimization for inter-layer parallelism on ResNet-18 ❧ and VGG16 ❧ for a multi-accelerator (S3) pipelined deployment. In the 1st state, we optimize for latency and identify the bottleneck layer (highlighted in bold), which determines the pipeline latency. In the 2nd stage, we optimize for energy (or power) by allowing the latency of other layers to increase, while staying less than the pipeline latency.

<table>
<thead>
<tr>
<th>Layer</th>
<th>ResNet 18</th>
<th>VGG16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1st stage</td>
<td>2nd stage</td>
</tr>
<tr>
<td>1</td>
<td>4.9E+04</td>
<td>5.0E+02</td>
</tr>
<tr>
<td>2</td>
<td>1.6E+05</td>
<td>1.5E-01</td>
</tr>
<tr>
<td>3</td>
<td>4.2E+04</td>
<td>4.2E-02</td>
</tr>
<tr>
<td>4</td>
<td>9.4E+04</td>
<td>5.6E-03</td>
</tr>
<tr>
<td>5</td>
<td>2.9E+04</td>
<td>5.6E-03</td>
</tr>
<tr>
<td>6</td>
<td>1.3E+04</td>
<td>1.5E+04</td>
</tr>
<tr>
<td>7</td>
<td>4.1E+04</td>
<td>2.0E+02</td>
</tr>
<tr>
<td>8</td>
<td>2.1E+04</td>
<td>5.8E-02</td>
</tr>
<tr>
<td>9</td>
<td>4.9E+04</td>
<td>5.0E-03</td>
</tr>
<tr>
<td>10</td>
<td>1.6E+04</td>
<td>1.0E+05</td>
</tr>
<tr>
<td>Max.</td>
<td>1.6E+05</td>
<td>2.0E+04</td>
</tr>
<tr>
<td>Ave.</td>
<td>5.0E+04</td>
<td>6.4E-03</td>
</tr>
</tbody>
</table>

We only display the layers with unique shape. Maximum and Average are calculated based on all 20 layers of ResNet-18 ❧. We display the summary of VGG16 for the interest of space.

5.4 Two-stage Optimization for Inter-layer

So far in this paper, we consider three systems: S1, S2, and S3, to parallelize the computation of a DNN layer, whose scenario can be termed as intra-layer parallelism. Next, we show how GAMMA can also be applied to the scenario of inter-layer parallelism. We consider a S3 system with inter-layer parallelism scenario, used in prior multi-accelerator systems [18, 46, 62], where each accelerator is handling one layer of a model, and the entire model is executed as layer-wise pipeline manner on the system.

5.4.1 Motivation. The pipelined system can often bring higher throughput. However, it also owns the problem of being bottleneck by critical block. The layer-wise pipelined accelerator can be bottlenecked by some computation-heavy layer. As the bottleneck latency exists and may not be able to be further optimized, in this case, we relax other non-critical blocks by relaxing their timing constraint to achieve overall lower energy/power of the system. 5.4.2 Structure. We apply a two-stage optimization method, where we optimize latency first and then power/energy at the second stage.

Stage I: optimize latency. We use GAMMA to find the mapping that optimizes the latency of each layer. We identify the bottleneck layer, whose latency decides the pipeline latency of the system.

Stage II: optimize power/energy. With the pipeline latency decided, we relax other layers by applying GAMMA again but optimizing power/energy at this stage with the awareness of not exceeding the pipeline latency. This is formulated by adding a heavy penalty when the searched solution exceeds the pipeline latency.

With the designed two-stage optimization, we can optimize the throughput of a layer-wise pipelined system at the first stage and further optimize its power/energy efficiency at the second stage.

5.4.3 Results. Table 6 shows the HW performance of each layer in ResNet-18 in the 2-stage optimization scheme. In the Latency-Power experiment, we optimize latency first and their power next. After the first stage, it shows that the latency is bottlenecked by the second layer, and it decides the pipeline latency. With the awareness of the pipeline latency, we optimize power at the second stage and find we could reduce the power by 95% comparing to the first stage when remaining at the same pipeline latency. The Latency-Energy experiment shows 58% reduction on energy consumption. Likewise, we execute the same flow on VGG16 and found it also effectively reduce the power by 99% and energy by 78% respectively.

6 RELATED WORKS

6.1 Dataflow Design in DNN Accelerators

Dataflow design has been a popular topic in the research of DNN Accelerators. Multiple hand-designed dataflows have been used across accelerators, categorized [10] as output-stationary [15, 19, 37], weight-stationary [1, 5, 6, 16, 36], row-stationary [10], input stationary, and no local reuse [7, 69]. In this work, we provide a framework to automatically determine an optimized dataflow and mapping. GAMMA can be used at compile-time to configure in the mapping if the underlying accelerator supports multiple dataflows [30, 33], or at design-time to determine the right dataflow for a custom accelerator developed for running a fixed set of DNNs.

6.2 HW Mapping Space Search and Exploration

Many recent works have been developed to tackle DNN HW mapping. However, since the search space is extremely large, many of them restrict the search space by considering only part of the aspects of the HW mapping search space. Some consider a limited combination of HW mappings and pick among them [33]. Some constrain the parallelizing dimension to a few choices [17, 18, 56, 63]. Some fixed the computation order to a subset of all combinations [47, 51, 62, 66, 68]. Some vastly reduce the space of tiling sizes [52] by a heuristic, or large step size, e.g., power of two [54]. Interstellar [14, 67] considers all three aspects of HW-mapping, but they constrain the search space by limiting the choice on each aspect such as the choices of loop order, parallelizing dimension. All these prior arts exclusively rely on exhaustive/random search with the help of coarse-grained striding enumeration or heuristics-based pruning. On the other hand, to search the mapping space with sample efficiency, Suda et. al [56] and TensorComprehensions [61] uses genetic algorithm, AutoTVM [8, 9] uses simulated annealing and boosted tree, Reagen et. al, [41] uses Bayesian optimization, RELEASE [4] uses RL to formulated a more guided search by ML technique. However, these ML-based algorithms need to work in a pre-defined rigid design space, where the level of parallelism is restricted, and hence the mapping space is constrained. The mappers in Timeloop [35] and Simba [46] explore the full search space; however, they rely on exhaustive/random search. In this work, we explore a full search space, but with a ML-based guided search method with sample efficiency.

7 CONCLUSION

Finding optimum mappings of DNNs on accelerators is critical for performance, but is challenging to automate due to an extremely large layer-specific and HW-specific search-space. In this paper, we propose a GAMMA, a genetic algorithm-based technique for the HW-mapping problem. GAMMA consistently outperforms other search techniques. With new DNN models and new accelerators being proposed at an unprecedented rate, GAMMA allows researchers to quickly explore the HW efficiency of emerging DNNs without time-consuming human-in-the-loop mapping and tuning processes.
REFERENCES

[15] Zidong Du et al. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In ISCA.