GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm

Sheng-Chun Kao
Georgia Institute of Technology
felix@gatech.edu

Tushar Krishna
Georgia Institute of Technology
tushar@ece.gatech.edu

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.

CCS CONCEPTS

• **Hardware → Hardware accelerators**.

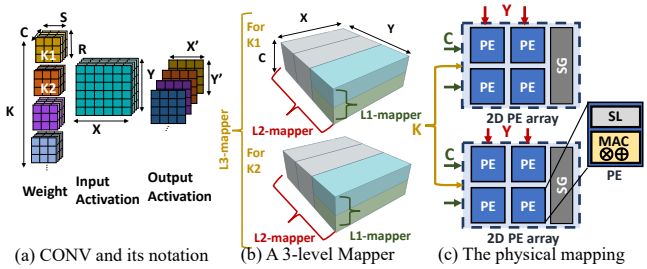
KEYWORDS

Genetic Algorithm, ML accelerator, Reconfigurable device

ACM Reference Format:
Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415639

1 INTRODUCTION

Deep neural networks (DNNs) are being deployed into many real-time applications such as autonomous driving, mobile VR/AR, and recommendation systems. However, DNNs are often strictly constrained by end-to-end latency or energy. This has opened up extensive research on computationally efficient DNN models [44, 59] and inference hardware accelerators [1, 10, 15, 25, 30].



(a) CONV and its notation (b) A 3-level Mapper (c) The physical mapping

Figure 1: (a) The six dimensions of a CONV operation. *K: output channels, C: input channels, Y: input height, X: input width, R: filter height, S: filter width, Y': output height, X': output width.* (b) A 3-level mapping example on input activations. Each level represents parallelism across a spatial dimension of the accelerator. (c) Physical mapping on accelerator. The L1-mapper parallelizes dim C across rows, L2-mapper parallelizes dim Y across cols, and L3-mapper parallelizes dim K across multiple arrays.

The architecture of DNN accelerators is determined by two key components: *HW resources* and *HW mapping strategy*. The HW resources (Fig. 1(c)) comprise of the total on-chip compute (hereby referred to as "PEs"), local scratchpad (SL) buffer in each PE, a global scratchpad (SG), and a network on chip (NoC) connecting them. The HW mapping strategy comprises of the tile sizes, computation order, and parallelization strategy (Fig. 1(b)). The design of computation order and parallelization strategy is also known as *dataflow* [10, 31]. The HW mapping and/or the HW resources are either fixed at design-time [1, 10, 15], or can be tuned at compile time (if the accelerator is reconfigurable, such as CGRA-based [30, 62] or FPGA [68]). The architectures are often designed based on the expected dimensions and shapes of the DNNs and heuristics. For example, the NVDLA [1] dataflow keeps weights stationary at PEs and parallelizes across input channels and output channels, optimizing for mid and late layers of many CNNs like ResNet [21] that exhibit this property. The Eyeriss [10] dataflow parallelizes across the activation and filter rows and keeps filter rows stationary.

Prior research [31, 35, 63, 68] has shown that there are no mapping strategies that can be efficient across all the layers of a DNN model. To exploit the benefit of different mappings, in this paper, we consider accelerators where the number of PEs and buffer sizes are fixed at design time, but the mapping can be configured at compile-time for each layer. The goal is to find the HW mapping for each layer of DNN models that optimizes the objective (min. latency/ energy).

Although multiple prior works [3, 14, 28, 29, 32, 34, 39, 53, 65, 68] have studied the mapping problem for DNN accelerators, the

This work was supported by NSF Award 1909900. We thank Hyoukjun Kwon for help with MAESTRO setup and feedback on the paper.

ICCAD '20, November 2–5, 2020, Virtual Event, USA

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 [33, 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 (**G**enetic **A**lgorithm-based **M**apper for **M**L **A**ccelerators). 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 optimizations methods[13, 20, 22, 23, 26, 38, 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 black-box 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.

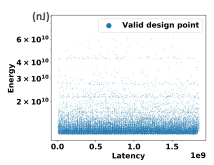
2.3 Mapping DNNs over Accelerators

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.

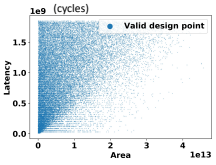
Computation order. 2D convolution, shown in Fig. 1(a), is a six loop-nest (or seven loops when considering batching). We can swap the order of these loops randomly, incurring 6! choices. Different orderings lead to different temporal reuse (*aka "stationary"* [10]) or spatial reuse (multi-casting) opportunities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



(a) Energy-Latency



(b) Latency-Area

	Min	Max
Latency (cycles)	8.18E+06	1.85E+09
Energy (nJ)	1.12E+10	8.34E+10
Power (mW)	1.52E+05	2.24E+09
Area (um^2)	1.77E+09	4.57E+13
SL size (Bytes)	2.62E+02	9.90E+06
SG size (Bytes)	2.82E+03	1.54E+07

(c)HW perf. statistics

Figure 2: The HW performance of randomly sampled HW-mapping in the design space on an example layer (second layer of VGG16). (a) The Energy to Latency plot, (b) The Latency to Area plot, and (c) The HW performance statistics of the sampled HW mappings.

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 $O(10^9) = 64^2 \times 224^2 \times 3^2$.

The tile sizes determine the number of operands of each tensor (input/weight/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 at design-time [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 space 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 increase 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], $(1 + \lambda)$-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).

Term	Description
Gene	The encoded value of one of the dimensions of a design point.
Genome (Individual)	A complete series of genes representing a deign point.
Elite	A set of genomes that has the highest evaluated fitness.
Population (Generation)	An entire set of genomes forms a population (one generation). The populations evolves with time by mutation/crossover and selection of the well-performing genomes to the next generation.
Crossover	Blend two parents' genes to reproduce children genomes.
Mutation	Randomly perturb a parent's genes to reproduce children genomes.

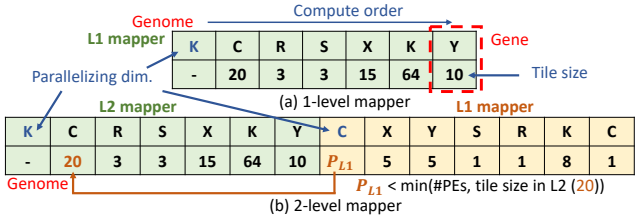


Figure 3: The GAMMA encoding example of (a) 1-level mapper and (b) 2-level mapper.

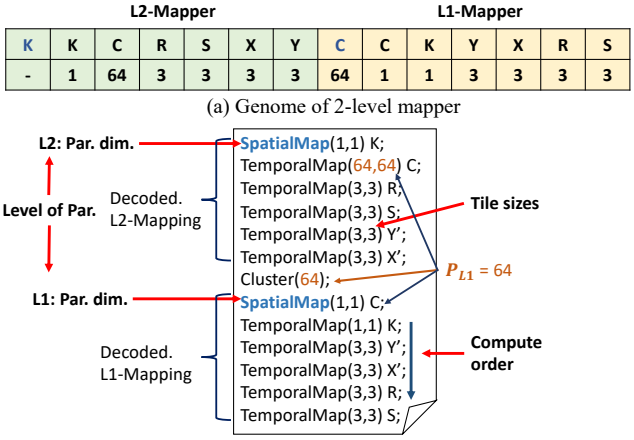


Figure 4: (a) GAMMA's description of a 2-level mapper and (b) its decoded description for cost model (MAESTRO) of a NVDLA-like [1] 2-level mapper.

3.2 Genetic Algorithms (GA)

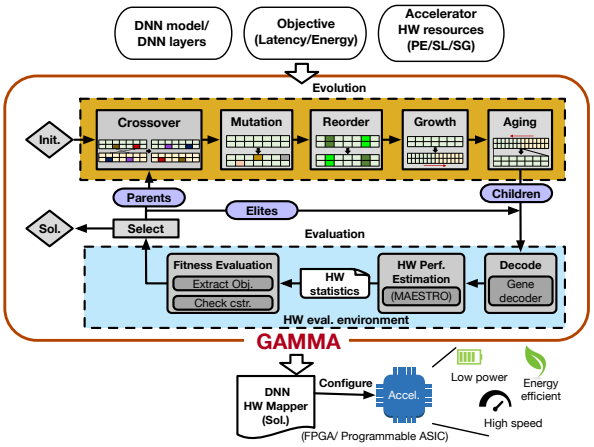
In this work, we develop a GA-based search technique. We list some common terminology for GA, namely *gene, genome, elite, population*, we will use across this paper in Table 1. A genome is a mapping solution in our context. We reproduce the next generation by mutation and crossover. The goal of GA is to retain well-performing genes across the evolution.

Benefits of GA. GA is one of the most popular algorithms for the scheduling problem for its lightness and simplicity [12, 24, 48, 50, 64]. Research shows GA reaches competitive performance with deep reinforcement learning [43, 55], and hyperparameter optimization problem. STOKE [45] and TensorComprehensions [61] use GA to search the space of DNN code optimization.

Challenges with standard GA. Standard GA still falls into the pits of the algorithm needing rigid input length. To this end, we develop a way to adopt GA to our problem by designing a novel evolution mechanism, which allows it to be flexible, without adding up immense overhead such as adding encoder or training an seq-to-seq model. We discuss these details next.

3.3 GAMMA Encoding scheme

We design a specific genetic encoding scheme for the HW mapping problem. For a 1-level mapper, we encode them into a (7, 2) dimensions of the genome, which contains 7 pairs of genes, as shown in Fig. 3(a). A pair of the gene contains a DNN layer tensor notation (e.g, K, C) and its tile size. The ordering of pairs reflects the computation order. The first pair of gene tells the parallelizing dimension. The 2-level mapper in Fig. 3(b) is encoded in the same manner. P_{L1} describes number of parallel L1-mappers, which is constrained



(a) Structure and algorithm flow

Optimization methods		Description	Mapping Space Aspects			
			Tile Strat.	Com. Ord.	Par. Dim.	Par. Level
Baseline Methods		Defined a ***Par. Level***. Encode ***Tile Strat., Com. Ord., Par. Dim.*** into fixed length inputs, and evolve them with algorithms.	✓	✓	✓	✗
GAMMA	Crossover	Randomly pick two genomes from the parent subset. Interchange the value of their tile size.	✓			
	Mutation	Mutate Dim.: Sample a new Par. Dim. Mutate Tile size: Randomly pick a gene pair, sample a new tile size.	✓		✓	
	Reorder	Randomly pick two gene pairs. Swap their order.		✓		
	Growth	Concatenate a randomly initiated L1-mapper at the tail.				✓
	Aging	Remove the L1-mapper at the tail.				✓
	Summary		✓	✓	✓	✓

(b) The summary of evolution in GAMMA

Figure 5: (a)The structure and algorithm flow of GAMMA, and (b) The summary of evolution in GAMMA.

by the number of available PEs (the number of PEs defines the maximum amount of parallelism.), and the corresponding tile size of the chosen parallelizing dimension (since we need at least one element to distribute into each parallelism unit). The L1-mapper describes the inner loop. The L2-mapper describes the outer loop, while containing P_{L1} number of instances of L1-mapper.

3.4 Decoding Genomes into a Mapping

We outline how we describe the three aspects of mapping space in the cost model, and show how the genomes from GAMMA are decoded into the cost model's description. Fig. 4(a) is a mapper description in GAMMA and Fig. 4(b) is its corresponding description in the cost model (MAESTRO). The order of K, C, X, Y, R, S from left to right in (a) and top to bottom in (b) reflect the computation order. The number paired with dimension in (a) and the number inside the bracket in (b) reflects the tiling size on each dimension. In MAESTRO, we note the parallelized dimension as *SpatialMap* and remaining dimensions as *TemporalMap*. Therefore we mark the first element (indicating parallelizing dimension) in (a) as SpatialMap in (b). A level of mapper in GAMMA can be translated as a *Cluster* in MAESTRO, in which *tiling strategy, computation order* and *parallelism dimensions* are fully described. We formulate multiple level of parallelism by concatenating the cluster. The L1 and L2 mapper are decoded into the bottom and upper cluster.

3.5 Algorithm Flow

Fig. 5 shows the flow of the GAMMA algorithm. We discuss the detail of each function block next. We first describe how we adopt the generic evolution operators, Crossover and Mutation, to the HW-mapping problem, and then introduce three additional evolution operators (Reorder, Growth and and Aging) 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 paralleize 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.

Table 2: The HW resources in different platforms.

	# PEs	SL size (on-chip)	SG size (on-chip)
Edge Platform	168	512B	108KB
Cloud Platform	65,536	4MiB	24MiB

Table 3: Three target systems (Accel's infrastructures).

System	Description
S1:Fixed 2D Accel	Accelerator with 2D PE array (row x col) with fixed aspect ratio. This fixes P_{L1} to number of rows, and level of parallelism to 2. Example: TPU, NVDLA.
S2: Flexible 2D Accel	Accelerator with 2D PE array with flexible aspect ratio. The PE array can be configured to flexible 2D shape, which relax the choices of P_{L1}. We also relax the level of parallelism to be 1 or 2. Example: Eyeriss, Eyeriss_v2.
S3: Scale-out 2D Accel	Accelerator scaling out 2D structure. Accelerator comprises multiple 2D PE array instances, enabling parallelism across PE arrays. The level of parallelism is either 2 or 3. L1 and L2 mappers map within and L3 mapper maps across the PE arrays. Example: Simba, Tetris.

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.

$$Fitness = \begin{cases} reward, \text{ if constraint met} \\ -Infinite, \text{ others} \end{cases} \quad (1)$$

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 cloud TPU[25]) and edge 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

Table 4: Baseline optimization methods.

Alg.	Description
RS	Random Search. We randomly sample design points and keep the best solution.
GA	Genetic Algorithm. We encode the design point into a series of genes. We evolve the genes by mutation and crossover. *We use mutation rate: 0.1, crossover rate: 0.1, in the experiment.*
DE	Differential Evolution. In DE, we mutate by the *difference vector* (*DV*). When mutation, we sample a parent and extract its difference on each dimension with another parent to form a *local DV* and with the best parent to form a *global DV*. Next, we sample another parent and mutate it by adding the weighted sum of found *local DV* and *global DV* to it. *We use weighting for local DV: 0.8, weighting for global DV: 0.8, in the experiment.*
(1 +λ)-ES	(1 +λ)-Evolution Strategy. For each parent we mutate it to reproduce λ number 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 use (1+1)-ES in the experiment, where one parent generate one child.*
CMA-ES	Covariance Matrix Adaptation-ES. We use covariance matrix of the elite group 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.*
TBPSA	Test-based Population-Size Adaptation. We estimate the trend of the population's fitness values. 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.*
PSO	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 ω: 1.6.*
pPort-folio	Passive Portfolio. We maximize diversity by spreading the risk broadly to multiple instances (optimizers). We launch K numbers of optimization methods, each experiencing 1/K 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.*

within prior accelerators: NVDLA-like [1] (parallelizing K and C dim.), Eyeriss-like [10] (parallelizing Y and R dim.), and ShiDianNao-like [15] (parallelizing Y and X dim). All three of them have *fixed* 2-level parallelism dimensions. We create custom mappings (i.e., dataflow + tile-size) by setting appropriate tile sizes that fit within the SL buffers for both the edge and cloud platforms .

GAMMA. We set the populations=200, generations=50, and the mutation/crossover rate and execution rate of other evolving functions as 0.5.

5 EVALUATION

5.1 S1: Fixed 2D Accel.

In Fig. 6(a), we run the baseline algorithms and GAMMA to search for the HW-mapping for each of the 20 layers in ResNet-18 with S1 setting (i.e., 2 levels of parallelism) and edge platform constraint. We record the best solution (lowest latency) of each algorithm after they execute 10K samples (most comparing algorithms converge after 10K samples). Fig. 6(a) shows the latency of each algorithm's solutions, where we also plot the corresponding latency when using fixed dataflow. We observe the baseline algorithms and the fixed-dataflows are with competitive performance. However, GAMMA can consistently find better solutions than both methods.

Valid solution and different platform constraints. The HW-mapping is invalid when its requirement of HW resources exceeds the platform constraint. Some methods cannot find any valid solutions that conform to the constraint after 10K sampling. Therefore some methods have no solutions (NAN) in some cases as shown in Fig. 6(a). We did not show the result of Random Search here, since it ends up finding no solution for most of the cases, which also infers the complexity of the search space (it cannot find a valid solution in 10K samples). When we have more HW resource budget as Cloud platform in Fig. 6(b), all baseline optimization methods can start to find valid solutions and optimize on them. This shows how the imposed constraints increase the complexity of the problem.

Despite the fact that the optimization methods fail in some cases, their solutions are competitive to manual-design ones (fixed-dataflow) when they succeed, as shown in Fig. 6(a)-(b). It shows the potential of automating the HW-mapping design process by 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× to 440× less latency in Edge platform and 153× to (1.3E+7)× less latency in Cloud platform.***

5.2 S2: Flexible 2D Accel.

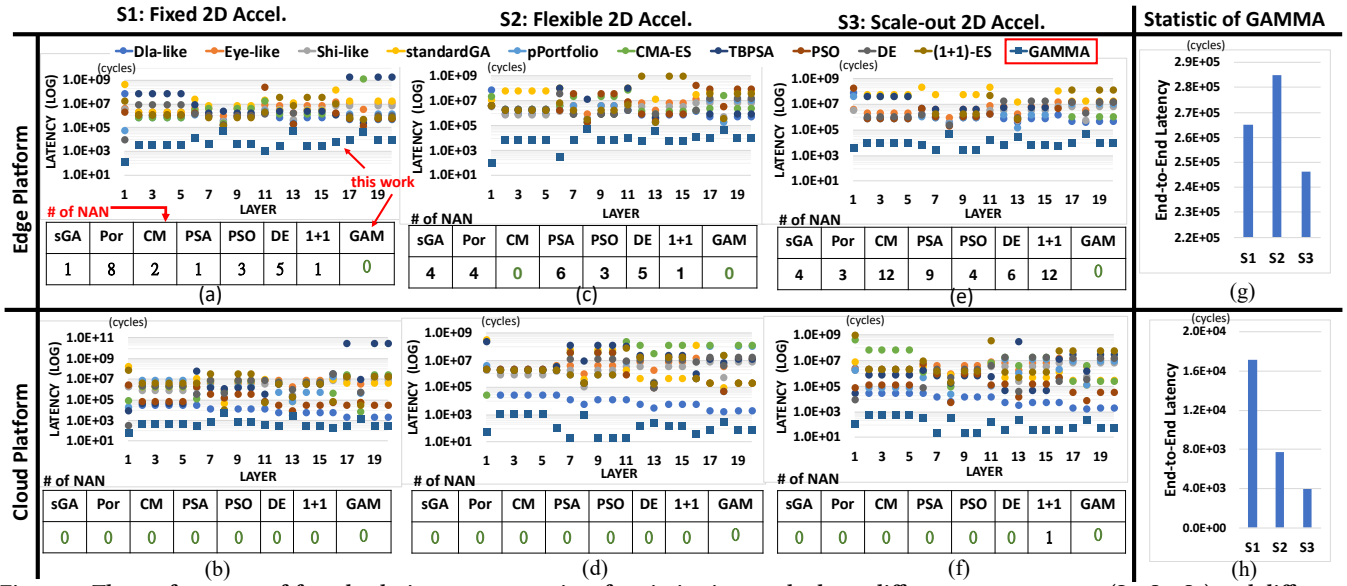
5.2.1 Objective: Latency. Fig. 6(c)-(d) compares latency for accelerators with flexible aspect ratios (i.e., the accelerator can support both 1-level and 2-levels of parallelism). One interesting observation is that the fixed ShiDianNao-like dataflow and NVDLA-like dataflow shows better performance than 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× to 1,035× less latency in Edge and 337× to (7.1E+5)× 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× to 36× less energy in Edge platform and 2× to 42× 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× to 644× less latency in Edge platform and 657× to (1.2E+5)× 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



	sGA	Por	CM	PSA	PSO	DE	1+1	GAM
(a)	1	8	2	1	3	5	1	0
(c)	4	4	0	6	3	5	1	0
(e)	4	3	12	9	4	6	12	0
(b)	0	0	0	0	0	0	0	0
(d)	0	0	0	0	0	0	0	0
(f)	0	0	0	0	0	0	1	0

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.

NAN: The method cannot find a solution that fits in the platform constraint within 10K samples.

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.

		MobileNet-V2				MnasNet				ShuffleNet				ResNet50			
Obj.	Accel.	GAMMA	NVDLA-like	Eyeriss-like	ShiDian Nao-like	GAMMA	NVDLA-like	Eyeriss-like	ShiDian Nao-like	GAMMA	NVDLA-like	Eyeriss-like	ShiDian Nao-like	GAMMA	NVDLA-like	Eyeriss-like	ShiDian Nao-like
Latency (cycles)	Edge	**9.67E+05**	1.46E+07	4.06E+07	7.23E+06	**8.31E+05**	1.43E+07	5.00E+07	8.44E+06	**5.20E+05**	8.41E+06	2.29E+07	3.89E+06	**6.49E+06**	1.04E+11	4.57E+08	1.31E+08
	Cloud	**1.85E+05**	9.33E+05	4.04E+07	6.04E+06	**1.39E+05**	4.03E+06	4.97E+07	7.46E+06	**3.41E+04**	6.28E+05	2.28E+07	3.37E+06	**3.84E+04**	2.91E+06	4.55E+08	1.13E+08
Energy (nJ)	Edge	**5.23E+08**	3.31E+09	9.62E+09	1.02E+10	**4.58E+08**	3.41E+09	9.707E+09	1.031E+10	**1.59E+08**	1.52E+09	4.647E+09	4.984E+09	**1.19E+09**	3.53E+10	1.1753E+11	1.245E+11
	Cloud	**4.12E+08**	8.13E+08	9.61E+09	1.02E+10	**3.72E+08**	7.71E+08	9.706E+09	1.030E+10	**1.17E+08**	2.57E+08	4.646E+09	4.978E+09	**1.13E+09**	2.14E+09	1.1751E+11	1.244E+11

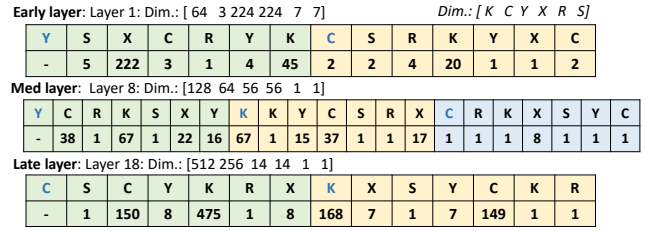


Figure 7: GAMMA's found mapping of early, medium, and late layer

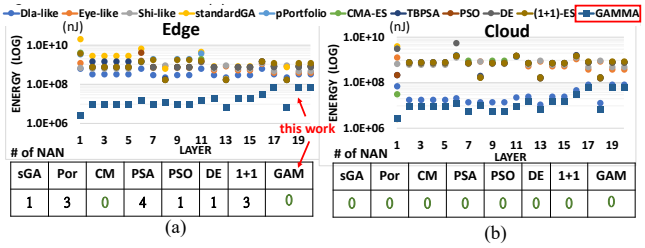


Figure 8: The energy consumption of S2 on ResNet-18.

along Y dim at L2-mapper. At the medium layer (Y=56, K=128, C=64, X=56), GAMMA found a 3-level mapper, which maps across Y, K, and C dimensions. At the late layer (K, C dominant, K=512, C=256), GAMMA parallelize C at L2-mapper and K at L1-mapper. From the above observation, we find the automatically evolved solutions are consistent with some heuristic and insight from the manual-designed dataflows [1] [1, 10, 15].

[1]The 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.

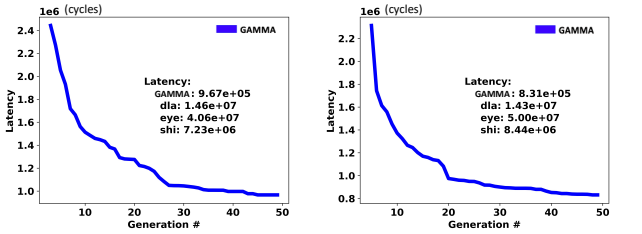


Figure 9: End-to-end latency improvement over generations with GAMMA for S3 system and edge platform constraint.

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 e.g., 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 always provides the lowest latency and energy. ***Across models and platforms, GAMMA finds solutions costing 5× to (1.2E+5)× less latency and 2× to (1.6E+4)× less energy***. Fig. 9 tracks how GAMMA converges to its solution across generations; this shows its sample efficiency via rapid improvement over generations.

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.

ResNet 18	Exp: Latency - Power				Exp: Latency - Energy			
	1st stage		2nd stage		1st stage		2nd stage	
Layer	Latency (cycles)	Power (mW)	Latency (cycles)	Power (mW)	Latency (cycles)	Energy (nJ)	Latency (cycles)	Energy (nJ)
1	4.9E+03	8.0E+02	7.1E+04	2.5E+02	1.1E+02	3.2E+06	3.8E+04	2.8E+06
2	**1.6E+05**	5.1E+02	1.6E+05	4.4E+02	1.0E+04	1.2E+08	5.2E+04	2.5E+07
6	4.2E+04	4.2E+02	1.3E+05	2.5E+02	2.9E+04	1.4E+08	6.1E+04	7.6E+07
7	4.6E+03	3.3E+02	6.5E+04	2.5E+02	8.9E+02	8.8E+06	2.4E+04	5.8E+06
8	4.9E+04	2.6E+02	1.5E+05	2.5E+02	**6.1E+04**	1.8E+07	6.1E+04	1.8E+07
11	2.9E+04	5.6E+03	1.5E+05	2.5E+02	2.6E+04	9.7E+07	3.7E+04	5.5E+07
12	1.3E+04	1.5E+04	1.4E+05	2.5E+02	2.2E+04	6.9E+07	4.2E+04	2.1E+07
13	4.1E+04	3.0E+02	1.5E+05	2.5E+02	2.5E+04	1.6E+07	5.1E+04	8.9E+06
16	2.1E+04	5.8E+02	5.9E+04	2.5E+02	2.3E+04	1.8E+08	3.7E+04	6.5E+07
18	4.9E+04	9.0E+03	1.5E+05	2.5E+02	5.5E+04	1.3E+07	5.9E+04	9.0E+06
19	2.6E+04	2.0E+04	1.4E+05	2.5E+02	1.0E+04	8.9E+07	3.8E+04	6.7E+07
Max.	**1.6E+05**	2.0E+04	**1.6E+05**	4.4E+02	**6.1E+04**	1.8E+08	**6.1E+04**	7.6E+07
Ave.	5.0E+04	**6.4E+03**	1.3E+05	**2.8E+02**	1.8E+04	**7.1E+07**	4.3E+04	**3.1E+07**
VGG16	Summary for Model VGG16							
Max.	**2.7E+06**	1.3E+05	**2.7E+06**	2.5E+02	**1.8E+06**	1.3E+09	**1.8E+06**	3.3E+08
Ave.	3.3E+05	**3.3E+04**	1.3E+06	**2.5E+02**	3.1E+05	**6.7E+08**	1.1E+06	**1.4E+08**

* 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 could 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 bottleneck 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

[1] 2017. NVDLA Deep Learning Accelerator. http://nvdla.org.
[2] 2020. MAESTRO tool. http://maestro.ece.gatech.edu/.
[3] Martín Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI 16*. 265–283.
[4] Byung Hoon Ahn et al. 2019. Reinforcement Learning and Adaptive Sampling for Optimized DNN Compilation. *arXiv preprint arXiv:1905.12799* (2019).
[5] Lukas Cavigelli et al. 2015. Origami: A convolutional network accelerator. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. 199–204.
[6] Srimat Chakradhar et al. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *ISCA*. 247–257.
[7] Tianshi Chen et al. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.
[8] Tianqi Chen et al. 2018. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*. 3389–3400.
[9] Tianqi Chen et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI 18*. 578–594.
[10] Yu-Hsin Chen et al. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*, Vol. 44. 367–379.
[11] Chen, Yu-Hsin and others. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *ISSCC*. 262–263.
[12] Ricardo C Corrêa et al. 1999. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed systems* 10, 8 (1999), 825–837.
[13] Christopher C Cox. 2017. A Comparison of Active and Passive Portfolio Management. (2017).
[14] Shail Dave et al. 2019. DMazerunner: Executing perfectly nested loops on dataflow accelerators. *TECS* 18, 5s (2019), 1–27.
[15] Zidong Du et al. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ISCA*.
[16] Clément Farabet et al. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Cvpr 2011 Workshops*. IEEE, 109–116.
[17] Mingyu Gao et al. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*. 751–764.
[18] Mingyu Gao et al. 2019. Tangram: Optimized coarse-grained dataflow for scalable NN accelerators. In *ASPLOS*. 807–820.
[19] Suyog Gupta et al. 2015. Deep learning with limited numerical precision. In *ICML*. 1737–1746.
[20] Nikolaus Hansen. 2006. The CMA evolution strategy: a comparing review. In *Towards a new evolutionary computation*. Springer, 75–102.
[21] Kaiming He et al. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.
[22] Michael Hellwig et al. 2016. Evolution under strong noise: A self-adaptive evolution strategy can reach the lower performance bound-the pccmsa-es. In *International Conference on PPSN3*. Springer, 26–36.
[23] John H Holland. 1992. Genetic algorithms. *Scientific american* 267, 1 (1992), 66–73.
[24] Edwin SH Hou et al. 1994. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed systems* 5, 2 (1994), 113–120.
[25] Norman P Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *ISCA*. IEEE, 1–12.
[26] James Kennedy et al. 1995. Particle swarm optimization. In *ICNN*, Vol. 4. IEEE, 1942–1948.
[27] Diederik P Kingma et al. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).
[28] Fredrik Kjolstad et al. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.
[29] Andreas Klöckner. 2014. Loo. py: transformation-based code generation for GPUs and CPUs. In *ACM SIGPLA Workshop on Libraries, Languages, and Compilers for Array Programming*. 82–87.
[30] Hyoukjun Kwon et al. 2018. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices* 53, 2 (2018), 461–475.
[31] Hyoukjun Kwon et al. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *MICRO*. 754–768.
[32] Nicholas D Lane et al. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *IPSN*. IEEE, 1–12.
[33] Wenyan Lu et al. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*. IEEE, 553–564.
[34] Yufei Ma et al. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *FPGA'17*. 45–54.
[35] Angshuman Parashar et al. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *ISPASS*. IEEE, 304–315.
[36] Seongwook Park et al. 2015. 4.6 A1. 93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications. In *2015 ISSCC Digest of Technical Papers*. IEEE, 1–3.
[37] Maurice Peemen et al. 2013. Memory-centric accelerator design for convolutional neural networks. In *31st ICCD*. IEEE, 13–19.
[38] Kenneth V Price. 2013. Differential evolution. In *Handbook of Optimization*. Springer, 187–214.
[39] Jonathan Ragan-Kelley et al. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.
[40] J. Rapin and O. Teytaud. 2018. Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad.
[41] Brandon Reagen et al. 2017. A case for efficient accelerator design space exploration via Bayesian optimization. In *ISLPED*. IEEE, 1–6.
[42] Ingo Rechenberg. 1994. Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. frommann-holzbog, Stuttgart, 1973. *Step-Size Adaptation Based on Non-Local Use of Selection Information. In PPSN3* (1994).
[43] Tim Salimans et al. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv* (2017).
[44] Mark Sandler et al. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*. 4510–4520.
[45] Eric Schkufza et al. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.
[46] Yakun Sophia Shao et al. 2019. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *MICRO*. 14–27.
[47] Yongming Shen et al. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *ISCA*. IEEE, 535–547.
[48] Pankaj Shroff et al. 1996. Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments. In *5th Heterogeneous Computing Workshop (HCW'96)*. 98–117.
[49] Karen Simonyan et al. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[50] Harmel Singh et al. 1996. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *5th IEEE heterogeneous computing workshop (HCW'96)*. 86–97.
[51] Linghao Song et al. 2019. HyPar: Towards hybrid parallelism for deep learning accelerator array. In *HPCA*. IEEE, 56–68.
[52] Mingcong Song et al. 2018. Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning. In *HPCA*. IEEE, 66–77.
[53] Michel Steuwer et al. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. IEEE, 74–85.
[54] Arthur Stoutchinin et al. 2019. Optimally scheduling CNN convolutions for efficient memory access. *arXiv preprint arXiv:1902.01492* (2019).
[55] Felipe Petroski Such et al. 2017. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv* (2017).
[56] Naveen Suda et al. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *FPGA'16*. 16–25.
[57] Ilya Sutskever et al. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.
[58] Christian Szegedy et al. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *21st AAAI conference on artificial intelligence*.
[59] Mingxing Tan et al. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946* (2019).
[60] Mingxing Tan et al. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2820–2828.
[61] Nicolas Vasilache et al. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).
[62] Swagath Venkataramani et al. 2017. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *MICRO*. 13–26.
[63] Swagath Venkataramani et al. 2019. DeepTools: Compiler and Execution Runtime Extensions for RaPiD AI Accelerator. *IEEE Micro* 39, 5 (2019), 102–111.
[64] Lee Wang et al. 1996. A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments. In *Proc. Heterogeneous Computing Workshop*. 72–85.
[65] Richard Wei et al. 2017. DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016* (2017).
[66] Xuechao Wei et al. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *DAC*. 1–6.
[67] Xuan Yang et al. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *ASPLOS*. 369–383.
[68] Chen Zhang et al. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA'15*. 161–170.
[69] Chen Zhang et al. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA'15*. 161–170.