AI\textsc{RCHITECT}: Learning Custom Architecture Design and Mapping Space

Ananda Samajdar
gatech.edu

Jan Moritz Joseph
RWTH Aachen University
Aachen, Germany

Matthew Denton
Georgia Tech
Atlanta, GA

Tushar Krishna
Georgia Tech
Atlanta, GA

Abstract—Design space exploration is an important but costly step involved in the design/deployment of custom architectures to squeeze out maximum possible performance and energy efficiency. Conventionally, optimizations require iterative sampling of the design space using simulation or heuristic tools. In this paper we investigate the possibility of learning the optimization task using machine learning (ML) and hence using the learnt model to predict optimal parameters for the design and mapping space of custom architectures, bypassing any exploration step. We use three case studies involving the optimal array design, SRAM buffer sizing, mapping, and schedule determination for systolic-array-based custom architecture design and mapping space. We perform systematic design-aware and statistical analysis of the optimization space for our case studies and highlight the patterns in the design space. We formulate the architecture design and mapping as a ML problem that allows us to leverage existing ML models for training and inference. We design and train a custom network architecture called \textsc{AI\textsc{RCHITECT}}, which is capable of learning the architecture design space with as high as 94.3% test accuracy and predicting optimal configurations which achieve on average (GeoMean) of 99.9% the best possible performance on a test dataset with \(10^5\) GEMM workloads.

I. INTRODUCTION

Custom architectures, in the form of accelerators, are arguably the de facto solution to create high performance hardware in the face of diminishing benefits from semiconductor device scaling. Among the many areas where custom architectures helped to meet the ever-increasing demand for compute performance, ML acceleration is probably the most noticeable example. As ML models start to become ubiquitous and more complex across various applications, their sheer demand for computing power and their inherent parallelism renders them too expensive and inefficient for general-purpose architectures; simultaneously creating a perfect case to reap the benefits of custom-designed accelerators [8]. Unsurprisingly ML accelerators are now used across all form factors from data centers to mobile phones, providing record-breaking performance and energy efficiency [5], [14].

Custom architecture design is a data-driven process. In a typical design cycle, the application characteristics are carefully studied for understanding possible optimizations, software bottlenecks, and computation/memory access patterns. Then multiple iterations of design space exploration (DSE) are performed for evaluating the costs of possible architecture choices via simulations or other cost models [15], [30]. Finally, based on the data obtained from DSE and applicable constraints, an implementation choice is made to get the best performance and efficiency.

In this paper, we demonstrate that the same data-driven decision-making can be learned by a ML model for the architecture design task. To get insights into the patterns, dimensionality, feasibility of learning the architecture design and mapping space we undertake three case studies. The first case study concerns the problem of finding the optimal shape and mapping strategy of a monolithic systolic array when presented with a general matrix multiply (GEMM) workload and design constraints. The second deals with sizing of SRAM buffer dimensions in a monolithic systolic array, when queried with the information about the buffer size limits, interface bandwidth, structure and mapping of the compute array, and the workload dimensions. The third case study explores the scheduling space of optimally running GEMM workloads on systolic arrays with different configurations.

With the help of our case studies, we systematically examine the optimization space of design and mapping to understand patterns and relationships between input parameters and optimal points, which can potentially facilitate learning of the space. Our analysis demonstrates that the architecture design and mapping space is learnable, at least within the limits of our use cases. We demonstrate that by formulating the optimization prediction as a ML classification, we achieve high degrees of accuracy in determining the optimal design parameters using a learned model. We design and evaluate a custom class of neural network architectures called \textsc{AI\textsc{RCHITECT}} that is capable of

Fig. 1. General structure of the proposed class of recommendation neural networks (\textsc{AI\textsc{RCHITECT}}).
learning the design space to get a prediction accuracy as high as 94.3% on a test dataset of 10^5 datapoints.

To the best of our knowledge, this work is one of the first that learns the design space to generalize it. Most prior works leveraging ML for accelerator DSE [4], [7], [16], [17], [27] focus on performing the search faster. Learning the design space enables constant time prediction of the optima. Such a feature has the potential to significantly speed up the conventional architecture design cycles. For instance, custom architecture design, by its very definition is tightly coupled with an application or an algorithm, which enables extracting orders of magnitude performance and energy efficiency as compared to general-purpose hardware. However, this tight dependence means that architecture design needs to be refreshed in tandem with changes to the application. For deep learning accelerator architectures, over the years, optimization objectives shifted from supporting GEMMs, to native convolutions [8], [21], and more recently to attention layers [13], [19]. For each such design cycle, an expensive DSE is needed to be performed repeatedly to find the architecture parameters corresponding to the new optima. A ML model similar to our proposal will significantly reduce the cost of such design iteration by condensing down the search for optimal configurations to constant time.

In summary, the following are our contributions:

- We present systematic, design-aware (Section III), and statistical studies (Section IV-C) on the design and mapping space of custom architectures via three case studies, demonstrating the feasibility of learning these spaces.
- We formulate traditional DSE tasks as ML classification/recommendation tasks by “quantizing” the optimization space and structuring the input and the output spaces. (Section IV) This abstraction enables different optimization tasks to be converted into learning problems, solvable by the use of existing ML classification models.
- We propose and evaluate AIRCHITECT (Figure 1), a custom-designed neural recommendation network capable of learning the DSE space to a high degree of prediction accuracy (Section V). The predicted optimal configurations generated by AIRCHITECT achieve 99.99% of the best achievable performance on a test dataset of 10^5 workloads when used for our case studies.

The paper is organized as follows. Section II defines each of our case studies and illustrate the complexity of the problem for finding the optima for a single query. In Section III, we generate multiple datapoints for each of the case studies and perform a design-aware analysis to identify underlying patterns across the datapoints which help with learning the space of optimum configurations. In Section IV, we cast the design-optimization as a learning problem, generate a training dataset, and perform statistical analysis. Finally we use both off-the-shelf and our custom designed ML model AIRCHITECT (Section V) to learn the design space and perform predictions. In Section VI we examine the existing literature, before concluding in Section VII.

![Monolithic systolic array template with programmable array dimensions, choice of dataflow, and configurable buffer sizes](image)

Fig. 2. Monolithic systolic array template with configurable array dimensions, choice of dataflow, and configurable buffer sizes

II. CASE STUDIES

In this work, we have contrasting requirements of choosing the design goal, which is complex enough to be challenging for naive design approaches but is feasible for human experts to reason about and validate the experimental results. We choose the problem of efficient systolic arrays-based GEMM accelerator design and mapping as our design-target for a ML model to learn. The simple construction and the easy scalability of systolic arrays creates a large design-space controlled via a few input parameters. Furthermore, a large abundance of literature for these structures fosters understanding the design space and examining the outcomes of the learning algorithms.

A. Background

**Systolic Array.** Systolic arrays are highly efficient structures for dense matrix multiplication and naturally have been adapted in many commercial and academic proposals for DNN acceleration [1], [2], [14]. Figure 2 depicts the schematic template for the target of our design task.

**Dataflow.** There are three distinct strategies of mapping compute, aka dataflows, onto the systolic array named Output Stationary (OS), Weight Stationary (WS), and Input Stationary (IS) [8]. The “stationarity” of a given dataflow is determined by the tensor whose element is not moved (i.e. stationary) for the maximum duration of time throughout the computation. Although many different dataflows exist for spatial arrays [20], we only consider true systolic dataflows that only use local (neighbor-to-neighbor) communication in this work.

B. Case Study Description

In this paper, we target three design-spaces / case studies:

- **Case Study 1: Array and Dataflow Prediction.** This case study focuses on the compute array in Figure 2. The array has two sets of design parameters we wish to learn and recommend for a given workload (i.e., DNN layer): (i) shape

![Diagram of systolic array](image)

Fig. 2. Monolithic systolic array template with configurable array dimensions, choice of dataflow, and configurable buffer sizes
of the array, determined by choosing the rows and columns, and (ii) dataflow.

– **Case Study 2: Buffer Size Prediction.** Given a compute array of a fixed size and shape, this case study focuses on sizing the SRAM buffers around it. As shown in Figure 2, the array is supported by three buffers. Two of the buffers prefetch and store the input operands (Input Feature map (IFMAP) and Filter in Convolution neural network terminology [8]), which are fed into the array from the top or left edges as determined by the choice of dataflow. The third buffer acts as temporary storage for the generated output elements (Output feature map (OFMAP)) or partial sums until they are drained.

– **Case Study 3: Multi-array Scheduling.** Given several heterogeneous arrays (with their own fixed sizes and memories), shown later in Figure 6, this case study focuses on scheduling multiple independent layers across them.

We motivate each case study in more detail next. We use SCALE-Sim [30] and some in house simulation tools to generate the data points (see Section IV).

**C. Compute Design Space**

For our systolic-array-based accelerator, the parameters in the compute design space are the shape of the array, and the mapping strategy or dataflow. In GEMM workloads, owing to the large parallelism and ease of scaling, the more computation elements (multiply-and-accumulate or MAC units) that are allocated to a problem the better. However, just providing the extra compute resources is not enough. The mapping and the shape of the systolic array determine the fraction of these compute resources that are utilized for work. Determining the best array shape and mapping has been widely covered in the existing literature. However, for the sake of our discussion, we perform some simple experiments to depict the complexity of the problem.

First, we depict the effect of the shape of the array when running the first layer of ResNet-18 [11] in Figure 3. Figure 3(a) shows three plots, each for the dataflow Output Stationary (OS), Weight Stationary (WS), and Input Stationary (IS), respectively. In each of these charts, we plot the runtime obtained for running the given layer for different array shapes possible when using $2^9$ MAC units, with each array having at least 4 rows or columns. In Figure 3(b), we depict the result for a similar analysis on this layer (first layer in ResNet-18) for the various array shapes when $2^{15}$ MAC units are used. First, we observe from this chart is that we can have orders of magnitude in difference in the runtime between a poor configuration and the optimal one. For example, for $2^{15}$ MACs using OS dataflow, using a shape $4 \times 8192$ takes 100× more cycle than when using the optimal configuration of $512 \times 64$. Second, we see that for a given shape of the array, the array utilization correlates with better runtimes. This is logical as, the higher the utilization is, the more parallelism can be exploited resulting in higher performance. However, utilization is not a sufficient condition for determining optimality. For example, when using OS dataflow with $2^9$ MAC units, all array shapes but 1 has full utilization ($8 \times 64$ to $128 \times 4$); however the runtimes among the configurations vary with the optimal configuration ($16 \times 32$) taking almost half the time than $128 \times 4$. When considering the case of $2^{15}$ units and IS dataflow, the optimal configuration ($64 \times 512$) has a lower utilization than 100% obtained by $4 \times 8192$ and still achieves a tenth in runtime.

We note that even with the same workload and dataflow, the optimal shape of the array changes with the design constraint itself. In Figure 4, we plot the array shape of the optimal point when the number of MAC units used changes from $2^5$ to $2^{15}$ for the first layer and eighth layer of ResNet18 [11].
Stalls due to memory

We observe that for a small number of OS dataflow, 32x32 array, 100 bytes/cycle BW (d) OS dataflow, sizes.

dimensions, external bandwidth, etc also determine the relative mapping/dataflow, other architecture parameters like the array
intuitive given the fact that several extraneous factors like operands themselves. This task becomes challenging and non-
be allocated out of this budget. It is well known that the capacities for buffers holding the three operands need to
the systolic-array-based accelerator depicted in Figure 2, the for compute, buffering, other components in the chip. For
typically an upper bound placed on the real estate available
the GEMM workload at a time, For each such unit, the dataflow is
g人才队伍 change, the optimal architecture and mapping config-
GEMM workload to an equal number
of systolic array-based distinct compute units, such that the overall runtime is minimized. Figure 6 helps describe the
problem qualitatively. In this particular example, we have
three compute units, one a monolithic 32 x 32 systolic array,
a distributed collection of 4 x 8 systolic arrays, and a fine-
grained distributed accelerator with 256 2 x 2 systolic arrays.
As evident by the example, the distinct compute units can be
set to the lowest size as all sizes lead to zero stalls, while
the resources of the capacity can be allocated for storing the
outputs. In Figure 5(b), we just change the dataflow from OS
to IS and notice a significant change in the stall behavior. In
this case, the only buffers that account for non-zero stalls are
the ones responsible for holding the output buffer elements (ie
the Output feature map or OFMAP). Consequently, the optimal
sizing allocates minimum buffer sizes for the input matrices
and the maximum to the buffer holding OFMAP elements.

Varying other architecture parameters like the interface bandwidth and the dimensions of the array also change the
optima. When the interface bandwidth is changed to 100
words/cycle while keeping the same dataflow (OS) and the
array dimensions (32 x 32), we observe that allocating 2048
KB to IFMAP buffer leads to zero stalls from IFMAP transfers,
as shown in Figure 5(c). In this case, given the stalls are then
only encountered from the OFMAP buffer, the configuration
with 1048 KB allocated to this buffer turns out to be optimum.
When using array dimensions of 8 x 128, the optimal is obtained
by allocating 1024 KB for the IFMAP and 2048 KB for the
OFMAP lead to the best configuration (see fig Figure 5(d)). The
lack of any apparent pattern and the dependence of the optimal
buffer sizing on the workload dimension and architecture
parameters limits the alternatives of optimization to iterative
search-based methods.

E. Multi Array Scheduling Space

In our third case study, we examine if the scheduling space of a heterogeneous accelerator can be learned. Although
the scheduling problem can come in many forms, for our experiments the specific problem we address of matching
a fixed number of GEMM workloads to an equal number
of systolic array-based distinct compute units, such that the
overall runtime is minimized. Figure 6 helps describe the
problem qualitatively. In this particular example, we have
three compute units, one a monolithic 32 x 32 systolic array,
a distributed collection of 4 x 8 systolic arrays, and a fine-
grained distributed accelerator with 256 2 x 2 systolic arrays.

We illustrate this by plotting the cost of improper allocation
in Figure 5. In this figure, we show the stall obtained due to
improper memory allocation on the second layer of GoogleNet,
when running on arrays with 2^10 MAC units with various
dataflows and external bandwidth. For this experiment, we
only allow the buffer sizes among 128 KB, 512 KB, 1024 KB,
and 2048 KB and estimate the stalls encountered. In Figure 5(a)
we depict the stalls encountered when the layer is run on a
32 x 32 array with OS dataflow, with each of the buffers backed
by a 50 words/cycle interface bandwidth link. For this case,
we observe that allocating the maximum amount of capacity
for the buffer containing Input Feature Map (IFMAP) buffer
leads to minimum stalls. The buffer containing Filter operands
can be set to the lowest size as all sizes lead to zero stalls,
while the rest of the capacity can be allocated for storing the
outputs. In Figure 5(b), we just change the dataflow from OS
to IS and notice a significant change in the stall behavior. In
this case, the only buffers that account for non-zero stalls are
the ones responsible for holding the output buffer elements (ie
the Output feature map or OFMAP). Consequently, the optimal
sizing allocates minimum buffer sizes for the input matrices
and the maximum to the buffer holding OFMAP elements.

Varying other architecture parameters like the interface bandwidth and the dimensions of the array also change the
optima. When the interface bandwidth is changed to 100
words/cycle while keeping the same dataflow (OS) and the
array dimensions (32 x 32), we observe that allocating 2048
KB to IFMAP buffer leads to zero stalls from IFMAP transfers,
as shown in Figure 5(c). In this case, given the stalls are then
only encountered from the OFMAP buffer, the configuration
with 1048 KB allocated to this buffer turns out to be optimum.
When using array dimensions of 8 x 128, the optimal is obtained
by allocating 1024 KB for the IFMAP and 2048 KB for the
OFMAP lead to the best configuration (see fig Figure 5(d)). The
lack of any apparent pattern and the dependence of the optimal
buffer sizing on the workload dimension and architecture
parameters limits the alternatives of optimization to iterative
search-based methods.

E. Multi Array Scheduling Space

In our third case study, we examine if the scheduling space of a heterogeneous accelerator can be learned. Although
the scheduling problem can come in many forms, for our
experiments the specific problem we address of matching
a fixed number of GEMM workloads to an equal number
of systolic array-based distinct compute units, such that the
overall runtime is minimized. Figure 6 helps describe the
problem qualitatively. In this particular example, we have
three compute units, one a monolithic 32 x 32 systolic array,
a distributed collection of 4 x 8 systolic arrays, and a fine-
grained distributed accelerator with 256 2 x 2 systolic arrays.
As evident by the example, the distinct compute units can be
a single or a distributed collection of arrays, working on one
GEMM workload at a time, For each such unit, the dataflow is
tunable at runtime. Hence, the task of the scheduler is to find
the match among each workload to the corresponding array,
when queried with three workload dimensions, and to choose the dataflows such that the overall runtime is the lowest.

This task is equivalent to the collection of several tasks like (i) identifying the workload on the critical path, (ii) finding the best compute unit and corresponding dataflow to run the critical path workload the fastest, (iii) performing the same analysis for the second, third, etc. slowest network in case of a tie among various possible scheduling combinations such that sum of time taken on all the compute arrays can be minimized.

The table in Figure 7(a) depicts the number of schedules possible for this simple example. Figure 7(b) depicts the runtime of the critical path for the first layer in GoogleNet, YoloTiny, and Alexnet.

III. DESIGN AWARE ANALYSIS

In this section, we analyze the datasets for our three design targets from an architect’s perspective to understand the design space and to highlight the trends and generalizations in the design space. We call this design aware analysis since we use the knowledge of the design space and the input architecture and workload parameter to understand the observed patterns.

A. Array shape and dataflow

In Section II-C, we discussed the complexity of determining the optimal array shape and dataflow for a given GEMM of the network on the critical path for all the possible schedules for the first layers in GoogleNet, YoloTiny, and Alexnet. The abrupt variation of runtimes for the neighboring schedules suggests that search is the practical way of determining to find the optima in this search space. Also, among the various workloads, the optimal schedule varies widely. For example, in Figure 7(c) we show a similar chart for the critical path runtime of third layers in the previous networks. The optimal schedule, in this case, is the one corresponding to entry 136 of the table in Figure 7(a), as compared to entry 10 in the previous figure. As with the previous case studies, predicting an optimal schedule is also challenging due to the lack of any apparent pattern, making it an interesting candidate for learning.
workload and resource constraints. However, we know that human experts can generalize to trim the search space or to find heuristics for making searches quicker with the knowledge acquired from experience. To further understand the design space and identify patterns for the optimum architecture and mapping configurations, we study the optimal design points obtained and we look into the solutions obtained by a search for a handful of workloads.

First, we examine the space of optimal array dimensions (height and width) obtained in Figure 8 when the design constraint (i.e., the maximum number of MAC units) is fixed to $2^9$. This chart is generated by searching the optimal for $10^4$ GEMM workloads, obtained by the sampling the operand dimensions ($M$, $N$, and $K$) from the distribution depicted in Figure 11(a) (see Section IV for details on dataset generation). In Figure 8(a) we show the optimal array dimensions for workloads that obtain the best runtime when using OS dataflow. The radius of the marker at each array dimension (row/col) captures the relative frequency at which the said dimension is deemed optimal. Figure 8(b,c) captures similar information for the workloads for which the optimal dataflow is WS and IS dataflow, respectively.

In Figure 9 we capture the relative frequency of optimal array dimensions for different values of maximum MAC units varied from $2^5$ to $2^{15}$. In each ‘circle’ of the figure, the optimal configurations for each dataflow (similar to the plot in Figure 8) is combined and presented using a different color. We observe that first the optimal array configurations always use the maximum number of MAC units, which is expected. Another observation, however, is that the optimal array configuration does not show any dependence on the optimal dataflow.

Figure 9 also helps in identifying the underlying pattern by combining the plots. The following observations can be made from this figure. First, we see a clear pattern that, the most frequent array configurations are the ones with are near square shape but have twice as many columns as rows, unlike the conventional approach of choosing square configurations [2], [14]. The square configuration of the array although optimal for a reasonable amount of workloads, does not work for most of them. Second, all of the array configurations are found to be optimal for at least one workload. Thus, none of the possible array dimensions can be summarily rejected from the search space to find the global optimum. Third, there is no apparent pattern that can be used for determining the optimal dataflow given the information about the optimal array configurations.

To understand the variation of optimal dataflow predicted, we study the design space for the shape of the operand matrices as depicted in Figure 10(a-c). For example, in Figure 10(a) the x-axis captures the ratio of height and weight of the first operand ($M:K$) capturing the shape of this matrix. The y-axis captures all the possible dimensions of the arrays using MAC units that range from $2^5$ to $2^{15}$ in powers of 2. Each data point represents the optimal array configuration and dataflow obtained by a GEMM workload, where the dataflow is represented by colors. The plots in Figure 10(b, c) depict similar distributions wrt to the shape of the second operand matrix ($K \times N$) and the output matrix ($M \times N$). One can observe from Figure 10(a) that distinguishing between OS and WS is possible just from the shape of the first operand matrix. However, determining either over IS dataflow is difficult as there exists a significant overlap. Similarly, Figure 10(c) depicts that the dimensions of the output operand matrix can help chose between WS and IS, while Figure 10(b) shows the same for IS over OS depending on the shape of the second operand.

### B. SRAM Buffer Sizing

Similar to the task of determining optimal array shape and dataflow for a given GEMM workload, given a constrained real estate the task of determining the optimal sizes of on-chip SRAM buffers is also challenging. In Section II-D we discussed the challenges involved in this task for a couple of example use cases. In a similar vein to our analysis in Section III-A, we try to understand if any generalizations or patterns exist in the design space of the on-chip memory buffer. For this study, we consider the optimal buffer dimensions in a systolic array, obtained by searching for $10^4$ distinct GEMM workloads running on randomly chosen array dimensions between $2^4$ to $2^{18}$, interface bandwidth between 1 to 100, and dataflow among OS, WS, and IS. For the sake of simplicity, we limit the search space to buffer sizes from 100KB to 900KB, changed in increments of 100KB. The objective of the search algorithm is to find the sizes of the three buffers simultaneously such that the overall stall cycles are minimized, and in case of multiple configurations with the same cost, the one with cumulatively minimum size.

Figure 10(d-f) depicts the correlation of the optimal buffer sizes, with the input parameters, interface bandwidth, the size of the workload matrices, and the dataflow. For the sake of clarity in all the figures, we plot the points corresponding to the minimum and maximum obtainable sizes. In Figure 10(d) we observe that, when using IS dataflow, small IFMAP buffers are the optimum. For WS dataflow, allocating the maximum buffer size appears to be compulsory. This observation makes sense when we consider the reuse patterns. In IS dataflow, the IFMAP matrix has the maximum reuse, whereas the elements of the filter operand matrix are streamed into the array, thus allowing a small buffer for the input to be sufficient. On the other extreme the WS dataflow requires the inputs to be streamed into the array, which requires a larger buffer size to lower the interface bandwidth pressure. Similar observation can be made in Figure 10(e) where the optimal sizes for the filter operand buffers are depicted. For this buffer, the smaller buffer dimensions are predominant when WS is used since the filter matrix has the maximum reuse. Similarly, larger buffer dimensions are obtained when IS is used. However, for the OS dataflow, where both IFMAP and Filter operands are streamed, larger buffers are favored by the IFMAP operand, while small buffers are common for the Filter operand. This trend can be explained by comparing the distribution of operand sizes of the matrices. The IFMAP matrix sizes vary from $10^6$ to $10^9$ bytes while the Filter operands vary from $10^5$ to $10^7$ bytes making the input matrices about two orders of magnitude larger. When
allocating from a fixed budget, it is perhaps optimal to allocate maximum memory to the larger matrix.

Figure 10(c) depicts the similar plot for the output buffer. In this chart, we observe that across the dataflow, small buffer sizes are optimum in general. This makes sense by considering the fact the OFMAP buffer does not contribute to extracting operand reuse and instead acts as temporary storage to hide the slow external interface. However, we notice that for large interface bandwidths and relatively small OFMAP sizes, larger OFMAP buffers are preferred. This observation can be explained by the fact that the optimal sizes are determined for all three buffers together. Furthermore, in this dataset we use the same interface bandwidth for all the buffers. When the bandwidths of the input interfaces are comparatively high, while the operand sizes are small (M and N) unless the K dimension is extraordinarily large, small input buffers are sufficient to obtain stall-free or low stall operation. Thus the stalls encountered from data draining contribute to a performance loss. At the same time, given there is a 'surplus' of allowable buffer, the size of the output can be increase to reduce the overall stalls encountered.

C. Multi-array Scheduling

This case study is a mapping problem similar to the optimal dataflow selection shown in Figure 10(a-c). The patterns observed in Figure 10(a-c) are almost similar to this case. For the sake of brevity, we directly show the statistical analysis for this case study in Figure 13(e)

IV. LEARNING ARCHITECTURE AND MAPPING SPACE

Our analysis in the previous section shows that the systolic-array-based accelerator design and mapping space possesses high-level patterns that indicate that it is perhaps possible to predict design parameters given that the distribution of the data is internalized. In this section, we discuss systematically structuring the learning problem, dataset generation, and briefly perform statistical analysis on the generated datasets.

A. Design Optimization as Learning Problem

The first step towards learning the observed patterns is to formulate the search-based optimization problems involved in our case study to machine learning-based regression or classification problem setting. Empirically we found classification tends to be a better fit for learning architecture and mapping optimization. A naive approach for employing classification for predicting architecture parameters is to independently predict each design parameter. However, there are a couple of problems with this approach. First, a separate model needs to be trained for each design parameter, which can get easily out of hand when dealing with a system with significant complexity. Second, the parameters are often inter-dependent, for example, the sizes of memory buffers in the systolic array, and different models might fail to capture the interdependence if trained separately. Motivated by these factors, we formulate the problem into a recommendation setting, where a bunch of parameters is clubbed into bins, and the trained model is expected to recommend the bin corresponding to the optimal parameters. This formulation helps us leverage the existing classifier models developed by the machine learning community and also helps us reason about the problem systematically.
As shown in Figure 12(b) each label serves as the index for a Distributed array schedule prediction case study the minimum array dimensions to be 2 dimensions that are powers of 2. In our experiments, we allow conventional systolic array designs, we only consider the array set of parameters, which for this case study denote the systolic in [1, 10]. The workload dimensions are determined by the distribution depicted in Figure 11(a). This distribution is generated from about 5 million data points, which took about a couple of weeks of times when running over several servers using a total of 800 logical CPU cores.

Case Study 2: Buffer Size prediction. In the second case study, our goal is to predict all optimum sizes of the three operand buffers in the systolic array (see Figure 2) simultaneously for given workload parameters, information about the compute, and design constraints.

The input space, as depicted in Figure 12(a), captures the maximum memory capacity, workload dimensions (M, N, K), the array dimensions, dataflow, and interface bandwidth of the buffers. For simplicity, we assume the same interface bandwidth for all three buffers. The limit for the maximum memory capacity is set to 3000 KB; the workload dimensions are sampled from the same distribution as the previous case study with limits observed in Figure 11(a). The array dimensions and dataflow are sampled from the output space of the previous case study to keep our experiments consistent. The interface bandwidth expressed in bytes/cycles is taken from the space of integers in [1, 100], sampled with uniform probability. The output space, is a list of labels, where each label indexes an entry of memory sizes for each of the buffers. The minimum size of each buffer is restricted to 100 KB, and the allowable sizes increment with a step size of 100 KB as well. The maximum allowable size is 1 MB for an individual buffer. With these constraints, the output space contains 1000 distinct points defining the search space. For this case study, the optimal memory size is the one that leads to a minimum or no stall. For different configurations with the same stalls encountered, the one with the smallest cumulative capacity is considered the best. For this case study as well, we use SCALE-Sim [30] to generate costs of the various memory sizes. We generate about 5 million datapoints, where a million point takes about a week when multiple runs are launched onto a server cluster totaling 800 logical CPUs.

Case Study 3: Multi-Array Scheduling. In Section II-E we describe the problem of scheduling a set of GEMM workloads on an equal number of a distinct heterogeneous collection of compute units. The input space, for this problem, is simply the GEMM workload dimensions (M, N, K) one for each compute array. The output space, is an id depicting the mapping of workload to the arrays and the corresponding dataflow to be used as shown in Figure 7(a).

It is worth noting that our case study concerns the schedules when the number and the dimensions of the arrays are fixed. This case study does not cover the cases where the number of arrays or the workloads changes. However, it is interesting to
We observe that when the number of arrays changes the possible number of scheduling strategies grows exponentially as well as combinatorially as depicted by the equation $N = 3^x \times x!$, where $x$ is the number of compute arrays.

Figure 11(b) depicts the growth of the space with the number of compute units. In this case study, we chose to learn the scheduling space of four arrays, which leads to 1944 possible entries in the output space. The decision to choose this configuration is purely pragmatic, intended to keep the dataset generation times bounded. The different arrays used in the study is shown in Figure 12(c).

Run time is used as the cost function for dataset generation. The schedule which leads to the least runtime of the slowest configurations have very few occurrences and are therefore captured 66%. Continuing, the ten most frequent configurations of the label categories for three case studies. The distribution is complicated and non-uniform, but structured.

The problem formulation discussed in Section IV, allows us to use off-the-shelf classifiers to capture the design space and make predictions about the optimal parameters for given workloads and design constraints. We test out various models with different degrees of complexity on the datasets generated for the three case studies. The table in Figure 14 show these models. To summarize we used the scikit-learn libraries implementation of support vector classifiers [10] with linear and radial basis kernel. The state-of-the-art tree boosting method called eXtreme gradient boosting (XGBoost [6]) available from the xgboost package [34]. We also implemented four multi-layer perceptron (MLP) networks in TensorFlow's Keras [3].

Figure 14 shows the accuracy obtained for the three case studies when $2 \times 10^6$ datapoints are used for fit/training. For the MLPs, the networks are trained for 15 epochs with 90:10 training-validation split. We used a categorical cross-entropy loss function with accuracy as the optimization objective. Among the various case studies, the one for memory size prediction is learned well among all the models, with MLP-B attaining about 63% validation accuracy. Support Vector Classification, however, was able to perform the best among the models considered attaining about 50% test accuracy for case study 1, 60% and 40% for the other two respectively.
We observe that the design space of case study 1 is learned with an embedding size of 16 and a 256-node MLP hidden layer. While the network generates a one-hot vector of length equal to the output space of the problem indicating the optimal design configuration, the performance of MLP-B vs AIRCHITECT at least by about 10% accuracy.

To gain further insights on the quality of training, we plot the distribution of the actual labels and the predicted configuration IDs for 10^5 previously unseen test data points for each case study. Figure 15(d-f) shows predicted vs actual distribution for the three case studies. The first observation is that the test datasets' actual distribution closely matches with distributions of the original dataset, shown in Figure 13(a-c) corroborating that the test set does not compromise on generality. Second, visually the predicted distribution for case study 1 almost perfectly matches the actual distribution, confirming that the network learned the design space. Third, we observe that the networks for case study 2 and 3, learn to predict the configurations with significant representation on the dataset, while the configuration IDs with low statistical probability is ignored as noise. The presence of large spikes in the case of memory size prediction biases the model, shown by the high frequencies of the top two configuration IDs (Figure 15(e)) leading to relatively low accuracy. Similarly, for case study 3, Figure 15(f) shows that the model was successful in learning the distribution of the large spikes. However, in doing so it ignored the configurations with lower frequencies, which in turn cumulatively lowered the accuracy figure. However, it is worth noting that the model ignoring the lower frequencies as statistical noise demonstrates that the model is robust and does not overfit for any of the use cases. Further improving prediction accuracy for different design spaces requires data engineering on a case-by-case basis and therefore, is out of the scope of this paper.

To understand the cost of misprediction within the use cases, we compute the performance (reciprocal of runtime) of the workloads in the test dataset, for the predicted and label configurations. In Figure 15(g) we show the performance of the predicted configurations normalized to the labels for the 10^5 datapoints. Due to the high prediction accuracy, we observe that only a few data points have catastrophic performance losses (>20% of the optimal) leading to a 99.99% average performance (Geometric Mean) of the best possible. In Figure 14, AIRCHITECT beats the best performing off-the-shelf classifiers at least by about 10% accuracy.
networks were not part of the training or validation dataset, but the model is able to predict the optimal array shapes and dataflow, when queried with a constraint of $2^{10}$ MAC unit limit. Interestingly for case study 3, which had relatively low accuracy, for most cases, the performance does not lead to catastrophic losses. Figure 15(h) shows that among the mispredictions, most of the points suffer about 10% to 15% loss compared to the best achievable, leading to an average of 99.1% of the best possible runtime on average (GeoMean).

VI. RELATED WORKS

ML for Architecture search: Apollo [35] is a recent work from Google, targeting sample efficient searching through the accelerator design space using reinforcement learning. Gamma [17] and ConfuciuX [16] are similar ML based architecture mapping and design space configuration search methods which use genetic algorithm and reinforcement learning (RL) respectively. AutoTVM [7] use ML model for cost prediction to improve fast mapping search during compile time.

ML for EDA: Recently there has been a significant push toward automating place-and-route using machine learning. Mirhoseni et al [23] use RL for task placement on a heterogeneous system. Wang et al [33] use GCN and RL for automatic transistor sizing. NVCell [28] is a RL based proposal from Nvidia to automate standard cell placement. Nautilus [25] uses genetic algorithm to improve FPGA place and route. Kwon et al [22], use online tensor-based recommender systems to aid place and route in chip design. To the best of our knowledge this work is the first work, which truly learns the architecture design and mapping space and "generalizes" to predict the optimal configuration in constant time without the requirement of searching the design space.

VII. CONCLUSION

This paper presents AIARCHITECT, a recommendation neural network for learning the architecture design and mapping space of systolic array based accelerators. To the best of our knowledge is the first work to show that architecture design space is learnable. Our approach significantly reduces the cost of design space exploration, by replacing expensive search with constant time recommendation of optima. The paper shows a systematic study of the design and mapping space in the context of design aware and statistical analysis, demonstrates the formulation of conventional search problem in to a machine learning problem, and finally proposes a model which outperform the existing off-the-shelf ML models to capture the mapping and design space by significant margin.

REFERENCES

[1] “Nvidia tesla v100 gpu architecture,” http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2018.
[2] “Xilinx ML Suite,” https://github.com/Xilinx/ml-suite, 2018.
[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,” 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[4] B. H. Ahn et al., “Reinforcement Learning and Adaptive Sampling for Optimized DNN Compilation,” arXiv preprint arXiv:1905.12799, 2019.

[5] S. Cass, “Taking AI to the edge: Google’s TPU now comes in a maker-friendly package,” IEEE Spectrum, vol. 56, no. 5, pp. 16–17, 2019.

[6] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, 2016, pp. 785–794.

[7] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Learning to optimize tensor programs,” in Advances in Neural Information Processing Systems, 2018, pp. 3389–3400.

[8] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 367–379, 2016.

[9] F. Chollet et al., “Keras,” https://github.com/fchollet/keras, 2015.

[10] XGBoost developers, “XGBoost Documentation,” https://xgboost.readthedocs.io/en/latest/index.html, version 1.3.3.

[11] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, “Device placement optimi-