Abstract—The value of flexibility in Deep Learning accelerators to adapt to diverse layer shapes and sizes is well-understood. Contemporary reconfigurable architectures depend on compilers or other components in the software stack for optimal configuration and mapping search to fully exploit the benefits of flexibility. In this paper we show that the configuration and mapping space of flexible accelerators can be learnt using machine learning by casting it as a classification or recommendation problem. The learnt model can be used to obtain the optimal configuration of the target accelerator in constant time without search. We propose ADAPTNET, a recommender system for obtaining optimal configuration and mapping for GEMM workloads running on a RECONFIGURABLE SYSTOLIC ARRAY (RSA). RSA is designed to be configured such that it can operate across a spectrum from a single monolithic array to a distributed collection of smaller arrays of various sizes with flexible aspect ratios. This allows us to simultaneously achieve scalability and high mapping flexibility while preserving operand reuse. ADAPTNET demonstrates 95% test accuracy compared to an exhaustive search to achieve optimum configuration, state-of-the-art classification techniques such as SVMs, XGBoost and MLPs. We also present, ADAPTNETX, a specialized core to run ADAPTNET in hardware. Together, RSA and ADAPTNETX enable us to demonstrate a new class of flexible accelerators which are capable of self-configuring in hardware for the given GEMM workload. We present a 32.768 TOPS instance called SAGAR that is capable of providing the same mapping flexibility as a compute equivalent distributed system while achieving 3.5× more power efficiency and 3.2× higher compute density demonstrated via architectural and post-layout simulation.

I. INTRODUCTION

Custom architecture design enables us to achieve high performance and energy efficiency for a given class of workloads in post Moore’s Law era. Highly specialized architectures however are inflexible to any variation in the nature of workload and thus can easily be rendered obsolete. To mitigate this limitation, there has been an increasing interest in developing flexible architectures which have additional components (interconnects, buffers, and configuration registers) to support changing workload requirements. For popular applications like DNN acceleration, several such flexible architectures have been proposed [10], [11], [12], [22], [37].

In all of the prior works on flexible DNN accelerators, however, the onus of finding and setting the best configuration lies on the software stack, typically using a compiler/mapper [14], [17], [28], [38]. This dependence causes a few deployment challenges: (i) a cost model has be to developed and integrated as an optimizer into the compilation stack to help find optimal

| Workload | Rigid Monolithic | Flexible Monolithic | Distributed |
|----------|------------------|---------------------|-------------|
| Utilization | Reuse | Optimal | Optimal |
| Reuse | No mapping search | No arch config search | No arch config search |
| Structural scalability | No mapping search | No arch config search | No arch config search |
| No mapping search | No arch config search | No arch config search | No arch config search |
| No arch config search | No arch config search | No arch config search | No arch config search |

Fig. 1. Comparison of scalability, utilization, and operand reuse in traditional monolithic and distributed accelerators, and the position of the proposed architecture mappings, without which the flexible design loses utility, (ii) an expensive configuration and mapping search has to be performed at compile-time before scheduling any workload. Usually mapping search in software is performed via exhaustive, heuristic or optimization algorithm-based approaches which take about a few seconds to hours [17], [28], [38], even with sophisticated ML assisted frameworks like autoTVM [4]. (iii) the search-time overhead also eliminates opportunities for deploying such flexible accelerators for domains/applications with soft or hard-real time inference targets.

In this work, we demonstrate that the mapping and con-
The constitution and interactions of the self adaptive (SA) and reconfigurable array (RA) components to make up the SARA accelerator called SAGAR in this work.

Fig. 2. The trade-off between improved runtime and lost operand reuse in compute equivalent monolithic and distributed systolic array configurations. (a) the theoretical minimum runtime, and the runtime obtained for stall free operation of monolithic and compute normalized distributed systolic array settings; and (b) the corresponding SRAM reads, normalized to theoretical minimum reads required when multiplying a $256 \times 64$ matrix with another $64 \times 256$ matrix.

The self adaptive (SA) unit of a reconfigurable accelerator can be learnt by a machine learning (ML) model, which can then be used to query for optimal parameters for any workload at constant time. Dependence on software stack can be eliminated by incorporating this learnt model into the hardware itself and querying it in runtime. We illustrate this via two contributions:

First, we design a scalable reconfigurable hardware optimized for GEMM workloads called RECONFIGURABLE SYSTOLIC ARRAY (RSA). RSA is developed upon the intuition that flexible accelerators often need to trade-off utilization, data reuse, and hardware complexity (i.e., scalability). This is illustrated in Figure 1. Rigid Monolithic arrays (e.g., TPU’s systolic array [15]), are simple to construct but offer no flexibility leading to high under-utilization for many workloads [30], [33]. Flexible Monolithic arrays (e.g., MAERI [22], Eyeriss v2 [6], SIGMA [30]) provide flexibility via clever use of interconnects and configuration logic, enabling high utilization for a majority of workloads. However, the increased hardware complexity hinders scaling, and the design requires external software support to exploit the benefits of reconfigurability [14], [17], [28]. Distributed architectures (e.g., Tangram [12], Simba [35]) help address the utilization challenge, since irregular workloads can be tiled on to these smaller arrays. However, this architecture leads to loss in spatial reuse (i.e., direct data-forwarding) that monolithic designs provide, and also requires data replication across the SRAMs of the individual arrays. Data replication leads to a decrease in overall on-chip storage capacity, leading to a loss of temporal reuse due to smaller tiles. Moreover, distributed arrays can exacerbate the mapping search problem [28], [35]. RSA aims to address the shortcomings of all three design strategies. It is a flexible accelerator capable of supporting mappings that can be realized by monolithic as well as distributed arrays by configuring to variable array dimensions and number of sub-arrays (as depicted later in Figure 5(d)), thereby enhancing both utilization and reuse. In practice, RSA closely approximates a flexible monolithic design, with a fraction of area cost.

Second, we present a systematic mechanism to cast the architecture configuration as a ML classification problem and discuss considerations for optimal model design, training, and performance of the model at inference. Specifically, we develop a custom ML recommendation system model called ADAPTNET that achieves a recommendation accuracy of 95% on a dataset of 200K GEMM workloads, and on average (GeoMean) 99.93% of the best attainable performance (Oracle). We also design a custom hardware unit to run ADAPTNET called ADAPTNETX. ADAPTNETX enables to get a recommendation response for any query in about 600 cycles which is at least about 6 orders of magnitude faster than software. Furthermore, ADAPTNETX consumes the same hardware real-estate and roughly the same on-chip memory capacity 1 for different arrays, thus proving to be a scalable solution in contrast to approaches like using configuration caches. With ADAPTNETX the configuration lookup using ADAPTNET can be performed at runtime, without involving the software stack.

Together, these two components enable us to develop a new class of accelerators that we call Self Adaptive Reconfigurable Array (SARA) (Figure 2). SARA accelerators can self adapt at runtime to optimized configurations for the target workload, without requiring compile-time analysis. We demonstrate an instance of SARA that we name ‘Shape Adaptive GEMM AcceleratoR (SAGAR)’ as shown in Figure 2 and evaluate its performance across various configurations. We show that SAGAR has 3.2× higher compute density and 3.5× improved power efficiency, over equivalent scaled-out systolic array. The extra flexibility costs <10% in area and 50% in power, compared to equivalent scaled-up systolic array. Compared to an area normalized state-of-the-art flexible scalable accelerator [30], SAGAR incorporates 45% more compute. When comparing compute-equivalent configurations, SAGAR consumes 43% less power and 30% less area.

II. RECONFIGURABLE ARRAY DESIGN

A. Motivation

To help understand the tradeoffs involved in choosing a performant configuration, and the associated loss of reuse we perform a simple experiment. We run one GEMM operation, involving operand matrices of sizes sizes $256 \times 64$ and $64 \times 256$ on various systolic array configurations. These are, a $128 \times 128$ monolithic array, and five distributed scale-out configurations viz. $4 \times 64 \times 64$, $16 \times 32 \times 32$, $64 \times 16 \times 16$ arrays, $256 \times 8 \times 8$ arrays, and $1024 \times 4 \times 4$ arrays. We obtain the runtime and memory accesses for running this workload on all the array configurations using SCALE-Sim [34] (see

---

1The only change in ADAPTNET between various RSA is the weight of the output layer, which is small in comparison to the embedding table which takes most of the on-chip space.

2means Ocean in Sanskrit, reflecting the shape flexibility of our accelerator.
we develop a flexible design obtained by augmenting a base

(i) Distributed arrays are more performant than an equivalent

configuration akin to a large single array or a collection of

smaller arrays, whenever required.

In Table I we inspect a few well known accelerator proposals

to categorize into various types introduced in Figure 1 viz. Rigid Monolithic (RM),

Flexible Monolithic (FM), and Distributed (Dist)

section V-A). In Figure 3(a) we show the runtime normalized

to the theoretical minimum required. Please note that

with the chosen matrix dimensions, the systolic arrays in all the

configurations are mapped 100% with useful computation. The differences in runtime in various arrays under 100% mapping

efficiency is attributed to the array filling and draining at each

serialization step (see sec III in [33]). We observe that the

configuration with 32 × 32 array is the most performant, beating

the monolithic configuration by about 2×. In Figure 3(b) we

depict the SRAM read accesses performed by all the array

configurations, normalized to the theoretical minimum number

of reads possible. From this figure we observe that the 32 × 32

configuration performs about 4× more memory accesses then

the monolithic. The excess memory accesses, which lead to

reduced energy efficiency, result from the loss of wire reuse.

From the discussion above we make two observations.

(i) Distributed arrays are more performant than an equivalent

monolithic array, even when mapping efficiency is 100% on

both. However, the optimal size of each device in a distributed

setting is workload dependent. (ii) Monolithic configurations

are strictly more energy efficient than distributed arrays, due
to loss the of spatio-temporal reuse in the latter.

In Table I we inspect a few well known accelerator proposals

to see scalability and potential to maximize utilization. We

notice that simple architectures that are easy to scale in size, 

under perform on extracting operand reuse. On the other hand, 

architectures with sufficient flexibility are not scalable. None 

of the architectures, including the ones with multiple arrays 

and NoC support, can create variable sized arrays or flexible 

array dimensions which can help simultaneously achieve high 

mapping efficiency and data reuse. In the next subsections 

we develop a flexible design obtained by augmenting a base 

monolithic systolic array with additional bypass paths along 

the row and columns. This design enables use to chose between 

configuration akin to a large single array or a collection of 

smaller arrays, whenever required.

B. Compute array

Traditional MAC units. In Figure 4(a) we show a tradi-
tional systolic array constructed by laying down Multiply-and-

accumulate (MAC) (Figure 4(b)) units in a 2D grid. Each 

MAC unit is designed to get an operand from either both 

(Left in, Top in) ports or from either of the ports, and perform 
multiplication and addition operation. In the next cycle the 
operand data received is sent to its neighbour over the peer-
to-peer links. The internal registers, and multiplexers enable 
the array to work in output stationary (OS), weight stationary 
(WS), and input stationary (IS) modes of operation [5]. This 
simple mechanism of data movement results in high wire reuse, 
but at the same time restricts the mapping of compute only 
to those operations which require same set of operands to be 
mapped along a row or a column.

Systolic Cells. The mapping flexibility in systolic arrays 
can be improved by allowing adjacent MAC units to work on 
different operands. To enable this, the architecture needs to 
provide for additional links from the SRAM to the MACs. 
Providing such links to each MAC however is costly in terms 
of area as well as energy since the spatial reuse over wires is 
compromised. To simultaneously achieve mapping flexibility 
and the advantages of spatial reuse in systolic arrays, we 
propose a design called systolic-cell. A systolic-cell is a small 
grid of traditional MAC units augmented with multiplexers at 
the edges. This enables them to chose the operands from the 
neighbouring MAC units or a separate set of operators available 
via bypass links. The MACs within a systolic-cell are connected 
using peer-to-peer links similar to that of a traditional systolic 
array. Figure 5(a) shows a 4 × 4 systolic-cell example. Please 
note that choice of the size of a systolic-cell is implementation 
dependent. In general, the smaller the cell size, higher the 
mapping flexibility, which comes at a cost of slightly increased 
area and power.

Scale-up and Scale-out using Systolic Cells. Larger arrays 
can be created by arranging and connecting the systolic-cells as 
depicted in Figure 5(b) using the peer-to-peer links. At the edge 
of each systolic-cell the muxes can be configured to connect to 
the bypass links. Please note that dedicated bypass links are 
allocated to each systolic-cell to allow concurrency. Attaining 
flexible mapping in such a design is a matter of configuring the 
multiplexers of the systolic-cells. Depending on the mapping , 
an user can chose not to use the bypass paths at all and 
use the entire array as a single monolithic unit by setting the
multiplexers to accept data only to/from the peer-to-peer links, (this is the case depicted in Figure 5(b)), which is equivalent to a scaled-up configuration. One the other hand, the user can set all the multiplexers to accept and deliver data solely to the bypass links, therefore operating as a cluster of arrays, each the size of a systolic-cell. This configuration, depicted in Figure 5(c) is equivalent to a scaled-out configuration. Figure 5(d) illustrates some of the possible configurations constructed using a 64 MAC units with 2 × 2 systolic-cells. As can be observed in this figure, not only can the array be configured to work in fully monolithic or fully distributed configurations, but also in any of the configurations in between. By setting the appropriate muxes in either pass-through or bypass modes, sub-arrays larger than systolic-cell size can be constructed (eg. 4 × 4, 8 × 4 etc in this example). Each of the sub-arrays have access to the scratchpad memory using the bypass links. Please note that when fully utilized, a larger systolic array improves energy efficiency over a distributed configuration of same number of systolic-cells, which is equivalent to a scaled-out distributed collection of systolic arrays. The bypass muxes are turned on to allow systolic-cells to directly connect to the SRAM ports which are all active. (d) Possible monolithic and distributed configurations possible in the reconfigurable Smart-Systolic Array (SSA) using 2 × 2 systolic-cells.

These requirements clearly dictate that high bandwidth bypass links are necessary. Another addition in our proposed architecture are the switches at the edges of the systolic-cells. However, these switches are simple multiplexers, which are configured statically for a given workload, without the need for any additional logic.

**Scalability via Pipelining.** On-chip wire scalability studies such as SMART [19] have shown that it is possible to traverse a few millimeters (9mm to 11mm) of wire length in 1ns before latching the signal. The authors in SMART achieved this using conventional asynchronous repeaters (a pair of inverters) placed 1mm apart. In RSA, repeated wires offer an opportunity to not only cross a single-systolic cell in a cycle, but in fact bypass multiple systolic cells within a single-cycle. In our reference architecture SAGAR, we perform place-and-route to determine the number of systolic cells per pipeline stage of the bypass links. At 28nm, we find that 8 systolic cells can be bypassed at 1GHz, as we demonstrate later in Section V-B, Figure 13(h). Note that pipelining the bypass links only adds a few cycles of fill time to the RSA, and does not impact the internal timing of the systolic array within each systolic cell (which is itself pipelined at each MAC unit).

**D. Scratch pad memory**

The array constructed from systolic-cells is backed by SRAM scratchpad memories, which are constructed as two individual buffers. Each of these buffers is dedicated to one of the operand matrices. Such scratchpad SRAM buffers are common in accelerators, and are designed to reduce the number of off-chip accesses and facilitate temporal reuse. Each operand buffer is operated in a double buffered fashion, so that the prefetch latency can be minimized. The system also contains a third buffer which is used to store generated outputs elements.
1. **recNetInference():** In this work we use a recommendation work in parallel. Writing select values to a register, whose individual bits drive systolic array like TPU, multiple control units are required to data movement. Please note that in contrast to a traditional GEMM operations to completion and orchestrate the required instance of systolic array controller is initiated to drive the partition.

2. Of the original operand arrays to be used by each individual logic, then partitions the original workload by marking portions of the GEMM computation.

3. The select lines. These configurations stay static throughout realize the partitioned configuration. This is accomplished by writing select values to a register, whose individual bits drives the select lines. These configurations stay static throughout the GEMM computation.

4. **partitionWorkload():** The control logic, then partitions the original workload by marking portions of the original operand arrays to be used by each individual partition. **systolicController():** Finally, for each partition, an instance of systolic array controller is initiated to drive the GEMM operations to completion and orchestrate the required data movement. Please note that in contrast to a traditional systolic array like TPU, multiple control units are required to work in parallel.

III. **Recommendation Model**

This section describes a neural network based recommendation unit which can simultaneously predict the optimal architecture configuration and mapping strategy, when a workload arrives. This system solves two problems. First it minimizes the changes required in a compiler for configuration and mapping search, thus easing deployment. Second, it enables real time reconfigurability. Given that a large reconfigurable array is most likely be deployed in data-center like use cases, the capability to adapt in real time will help achieve improved resource allocation and consequently meeting tight service-level-agreements (SLA).

A. **Architecture design as ML problem**

To facilitate learning the design space we have to frame the search problem into a ML task framework like classification or regression. We found that framing this as a classification or recommendation task works the best. This abstraction lets us leverage the existing works and models which have been invented by the ML community. An important step in solving this problem is to define the output space of the model. It is natural to assign bins for each of the design parameters and independently predict the optimal values for each parameter of interest. However, this would require a separate model to be trained and queried for each design parameter. We show that multiple parameters can be combined into a single output class and consequently can capture the design space using a single model.

In our case, the output space comprises of (i) The number and logical layout of the partitions, (ii) The dimensions of the arrays in each partition, and (iii) the mapping/dataflow to be used eg. output stationary (OS), weight stationary (WS), and input stationary (IS). Figure 7(b) shows this output space captured as categories of architecture configurations, indexed by the class ID. The learned classifier is expected to select an architecture configuration and corresponding mapping strategy which provides the optimal performance for the workload. To better visualize the complexity of the design space, in Figure 7(c) we depict the runtime and energy consumption for computation and SRAM reads when running layer 19 of FasterRCNN (see Section V-A) for the different architecture configuration and corresponding mapping strategy, when a workload arrives. This system solves two problems. First it minimizes the changes required in a compiler for configuration and mapping search, thus easing deployment. Second, it enables real time reconfigurability. Given that a large reconfigurable array is most likely be deployed in data-center like use cases, the capability to adapt in real time will help achieve improved resource allocation and consequently meeting tight service-level-agreements (SLA).

B. **Recommendation Neural Network**

**Dataset generation.** We generated a dataset of about 2 million workloads, by sampling M, N, and K dimensions from a uniform distribution of positive integers $\leq 10^4$. For each such workload dimension we searched through the configuration space of the reconfigurable array design using $2^{12}$ MAC units.
Accuracy (%)

(a) Size of the configuration space wrt number of MAC units for a systolic-cell based flexible array (b) Example of configurations predicted by ADAPTNET indexed by category ID (c) Performance and energy consumption by various configurations when running layer 19 of FasterRCNN (d) Chart showing the name and description and name of the various classifiers used in this work (e) The accuracies obtained by classifiers on predicting the architecture parameters for out dataset of RSA configurations with 2\textsuperscript{14} MAC units (f) Architecture of our proposed recommendation network

### Choosing the classifier.
Abstracting the problem in the form of a classification naturally opens up the choice of using existing classification algorithms. We explored a handful of pre-existing classifiers, some of which are listed in Figure 7(d). The Support Vector Classifiers and the XGBoost models we use are standard implementations provided in scikit-learn \cite{29} and xgboost [3] python packages respectively. We implement the MLPs in keras subpackage in tensorflow and train them for 20 epochs. In Figure 7(e), we show the prediction accuracy of these models on a test set of 200K points, after the model has been trained on 90% of the dataset. It is interesting to observe that among all the models only XGBoost was able to reasonably learn the design space and achieve about 87% prediction accuracy.

### Recommendation Model.
The performance of XGBoost model is encouraging and demonstrates that the design space can be learnt. To further improve the prediction performance of the model, we hand designed a recommendation neural network. We take inspiration from typical neural network based recommendation systems like DLRM \cite{26}, which is constructed by augmenting embedding lookups with MLP based classification. The presence of trainable embeddings help in mapping the input data from the raw input space to a latent space, which is observed to improve the classification performance. Given our use-case, there are two main requirements we need to satisfy. First, we need our network to have high accuracy in predicting the best runtime configuration which maximizes performance. Second, given that the recommendation network needs to be queried at runtime, the network should be small keep the inference latency and implementation costs low. In our use case, the recommendation inference for a given layer is run concurrent to the execution of a previous layer whenever possible. Lower inference latency therefore moves the recommendation step out of the critical path. Moreover, a smaller network has fewer computation and storage requirements and hence minimizes the overheads. Honoring these requirements, we propose a network as depicted in Figure 7(f). The network, called ADAPTNET, is simple, where we lookup the embedding entries for the input features, and then use a classifier with single hidden layer with 128 nodes and softmax activation at the output.

### Training, Performance, and Generalizability.
To train our recommendation network we use one Titan RTX GPU with 84 SMs. When training on the dataset for 2\textsuperscript{14} MAC based RSA, for 30 epochs with a mini-batch size of 32, it takes about an hour to converge. Figure 8(a) shows the accuracy progression as the training proceeds. We obtain a high accuracy of 95% of the test dataset of 200K points, which is compared against other classifiers in Figure 7(e). We also test the robustness of our design by generating similar datasets of 2M points each for RSA’s with varying number of MAC units (eg 2\textsuperscript{12}, 2\textsuperscript{13} etc). The aim is to test the performance of different ADAPTNET with different output configuration space. In Figure 8(b) we plot the test accuracies obtained for each such ADAPTNET trained for 30 epochs with 90:10 training-testing split. Please note that the data points in test datasets are unknown at training time. We observe that the networks all achieve high accuracies over 90%. To distinguish the ADAPTNET’s among themselves we use the size of the configuration space as a suffix. For example, the design space of 2\textsuperscript{14} MAC has 858 possible configuration,
Performance wrt \(2048\) and \(1024\) 1-D unit hardware for A which recommend the optimal configurations. The RA unit 10, however, this choice will lead to either fewer MAC units left for A. Hardware to run A Reconfigurable Array (SARA). SA, to ensure optimal performance. We believe this results recommendation to any query having workload dimensions owing to learned parameters, can generalize configuration it a non scalable solution. One the other hand, A configuration space of the cache, search has to be performed at runtime. The large C. Alternatives to A therefor we call the corresponding network A configurations predicted by A function of number of multipliers. (b) Architecture of the custom (a) Cycles needed to run A test samples when compared to the runtime of best possible ELF. A s and on the custom hardware unit (A ET) leading to an additional overhead. An alternative alternatives to ADAPT NET. We found a custom design tuned for ADAPT NET layer parameters to be more efficient. For efficient execution of the dense layers, we chose a 1-D multiplier unit with a binary tree based reduction as shown in Figure 9(b). We found Input stationary (IS) dataflow to be the most performant for our use case. In this mapping the elements of the input vector is buffered near the multipliers, while elements of the weight matrix are streamed through to generate one output element/partial sum, with a sustained throughput of 1 element per cycle. Throughput can be further increased by adding more such 1-D units. We name the custom core with one or more such 1-D units as ADAPT NETX. In Figure 9(a) we depict the variation of runtime of ADAPT NET inference on ADAPT NETX with two 1-D units as a function of multipliers. We find the 512 multipliers result in best runtime of 576 cycles, when running ADAPT NET for \(2 \times 14\) MAC unit \textit{systolic-cell} design. We also examine the cost of misprediction of ADAPT NET in Figure 9(c), where we plot the runtime of the predicted configurations from ADAPT NET-858 normalized to best possible runtime. We see that most mispredictions are benign and only a few misprediction lead to catastrophic performance losses, leading to a geometric mean of 99.93% of the best possible performance.

IV. SELF ADAPTIVE RECONFIGURABLE ARRAYS

By coupling ADAPT NET with a reconfigurable array, we can create a self adaptive system which can be conceptually viewed as a combination of two units, a Self Adaptive unit (SA), and a Reconfigurable Array (RA) unit as shown in Figure 2. The SA unit encompasses the software and hardware components which recommend the optimal configurations. The RA unit is the hardware unit capable of flexibly configuring to the recommended configurations and hence run the workloads. It is worth pointing out that this design class is not specific to a reconfigurable core for running GEMM workloads. Instead any Coarse Grained Reconfigurable Array (CGRA) unit, configurable at runtime, can be augmented with a suitable SA, to ensure optimal performance. We believe this results in a new class of designs, which we name Self Adaptive Reconfigurable Array (SARA).

A. Hardware to run ADAPT NET

In the context of our use case, an intuitive option is to allocate a few \textit{systolic-cells} from the main array to run ADAPT NET. However, this choice will lead to either fewer MAC units left for the actual workloads, or to allocate additional \textit{systolic-cells} for ADAPT NET leading to an additional overhead. An alternative to adding more \textit{systolic-cells} will be to add a custom hardware dedicated for running ADAPT NET. We explore both the \textit{systolic-cell} and custom hardware options below for ADAPT NET-858.

ADAPT NET Runtime on \textit{systolic-cells}. Figure 9(a) shows the cycles required for a single inference of the ADAPT NET as a function of multipliers used in \(4 \times 4\) \textit{systolic-cell} based array. Understandably, the runtime decreases proportional to the increase in number of multipliers as we increase the number of \textit{systolic-cells}, achieving the best runtime of 1134 cycles when using 1024 multipliers or 64 cells. When both the workloads and the recommendation engine is run on a same array; for a TPU equivalent machine with \(2^{14}\) MAC units, about 6.25% of the array needs to be allocated for running the ADAPT NET. Another choice could be allocating more hardware resources in terms of extra 64 \textit{systolic-cells} dedicated to run the recommender network. However, given that ADAPT NET has exclusively dense layers processing the embedding lookups, a systolic execution turns out to be sub-optimal.

ADAPT NET Runtime on ADAPT NETX. We found a custom design tuned for ADAPT NET layer parameters to be more efficient. For efficient execution of the dense layers, we chose a 1-D multiplier unit with a binary tree based reduction as shown in Figure 9(b). We found Input stationary (IS) dataflow to be the most performant for our use case. In this mapping the elements of the input vector is buffered near the multipliers, while elements of the weight matrix are streamed through to generate one output element/partial sum, with a sustained throughput of 1 element per cycle. Throughput can be further increased by adding more such 1-D units. We name the custom core with one or more such 1-D units as ADAPT NETX. In Figure 9(a) we depict the variation of runtime of ADAPT NET inference on ADAPT NETX with two 1-D units as a function of multipliers. We find the 512 multipliers result in best runtime of 576 cycles, when running ADAPT NET for \(2 \times 14\) MAC unit \textit{systolic-cell} design. We also examine the cost of misprediction of ADAPT NET in Figure 9(c), where we plot the runtime of the predicted configurations from ADAPT NET-858 normalized to best possible runtime. We see that most mispredictions are benign and only a few misprediction lead to catastrophic performance losses, leading to a geometric mean of 99.93% of the best possible performance.

B. SAGAR Accelerator

SAGAR is constructed by augmenting the \(2^{14}\) MAC RSA unit, laid out as \(32 \times 32\) grid of \textit{systolic-cells}, with ADAPT NETX running ADAPT NET-858 (see Figure 10). We chose this configuration as it has the same compute as the TPU v2, and the \(4 \times 4\) \textit{systolic-cell} size works the best for our workloads (see Section V-A). Since each row and column in this configuration has 31 bypass links and one link to MAC, each buffer is constructed as a collection of 1024 1KB banks.

Real-time Reconfiguration. The ADAPT NETX uses an additional SRAM bank of 512KB to store the embedding table and the weight matrices for ADAPT NET-858. Each configuration corresponds to a 3968 bit vector which sets the bypass muxes, once the layer is ready to be mapped.
language modelling network, and DNNs for reinforcement learning respectively. Figure 11(f-g) shows our sensitivity analysis using a few other well known networks.

**Baselines.** We chose a 128 × 128 monolithic systolic array and distributed array of 1024 × 4 × 4 arrays as our baselines as depicted in Table III. Both the arrays have same number of MAC units as TPUv2. Each array in distributed configuration resembles the tensor cores in Nvidia GPUs. Both that baselines have the same total SRAM memory capacity of 3MB divided into buffers for staging two operand and one output matrix.

**Performance Analysis.** We model both of the baseline systems and SAGAR in SCALE-Sim and compare the performance for our workloads. In Figure 11(a) we depict the cycles taken to run all the layers in AlphaGoZero, DeepSpeech2, and the first 10 layers of FasterRCNN networks. Among the baselines, the distributed configuration mostly results in faster runtime owing to higher mapping flexibility. However SAGAR, owing to reconfigurability is capable of matching the better baseline configuration. Naturally, this flexibility leads to lower aggregated runtime for SAGAR than either of the baselines. We see this trend generalizing in Figure 11(f) as well.

**systolic-cell Design Space Exploration.** SAGAR is also capable of realizing configurations which are out of scope of either of baselines. This allows SAGAR to achieve higher performance than both the baselines on certain layers. For example, consider the synthetic GEMM operands depicted in Table IV. Figure 12(a) depicts the histogram of the best configuration for these layers obtained from simulation. The layers favouring 8 × 8 or 32 × 32 configurations constitute about 40% of the set. Neither of these configurations can be realized a fixed array configuration like the baselines. In Figure 12(b,c,d) we show the histogram of a similar experiment conducted on our DNN workloads. For these specific workloads, the 4 × 4 configuration works the best for majority of the layers. This observation also explains our findings in Figure 11(a) on why SAGAR’s performance is identical to the 4 × 4 baseline. Nevertheless, for layers which favor configurations like 8 × 8, 32 × 32 etc. SAGAR will lead to lower runtime than both the baselines. This is depicted by Figure 11(c), where we see that SAGAR achieves about > 10× speedup over monolithic when distributed configurations are preferred. While in cases where monolithic is preferred it runs faster than both the baselines.

**SRAM reads and Energy efficiency.** In general, due to the loss of reuse, distributed configurations with smaller array sizes have more SRAM reads resulting in lower energy efficiency. We observe this trend in action in Figure 11(b) where we depict the number of SRAM reads performed for layers when running our

---

**TABLE III**

| Name | Num Units | MAC/unit | Banks per SRAM buffer | Capacity per SRAM bank |
|------|-----------|----------|-----------------------|------------------------|
| Dist. 4x4 units (Baseline) | 1024 | 16 | 4 | 2.56 MB |
| Dist. 8x8 units | 256 | 64 | 8 | 512 B |
| Dist. 16x16 units | 64 | 256 | 16 | 1 KB |
| Dist. 32x32 units | 16 | 1024 | 32 | 2 KB |
| Dist. 64x64 units | 4 | 4096 | 64 | 4 KB |
| Monolithic 128x128 (Baseline) | 1 | 16384 | 128 | 8 KB |
| SAGAR | 1 | 16384 | 1024 | 1 KB |

---

**TABLE IV**

| Dimensions for the synthetic GEMM workloads | G1 | G2 | G3 | G4 | G5 | G6 | G7 | G8 | G9 | G10 |
|---------------------------------------------|----|----|----|----|----|----|----|----|----|----|
| M 128 256 512 1024 2048 | 64 | 64 | 64 | 64 | 64 | 64 | 64 | 64 | 64 |
| K 128 256 512 1024 2048 | 64 | 64 | 64 | 64 | 64 | 64 | 64 | 64 | 64 |
| N 128 256 512 1024 2048 | 64 | 64 | 64 | 64 | 64 | 64 | 64 | 64 | 64 |

---

**V. Evaluations**

We evaluate SAGAR in two settings. To capture the merits of the architecture, we present results obtained from simulation. While the implementation aspects are captured by reporting PPA number obtained from Place-and-Route (PnR).

**A. Architectural evaluations**

**Methodology.** For our architecture level studies we chose to use SCALE-Sim [34]. SCALE-Sim is a cycle accurate simulator for systolic array, which generates per cycle data accesses to and from various memories. This enables us to estimate and compare performance, energy consumption, power etc. of systolic array based components to a certain degree of accuracy. We created in-house scripts to generate SCALE-sim input files to perform the workload partitioning for the configurations recommended by ADAPTNET-858.

**Workloads.** For our evaluations we choose FasterRCNN [31], DeepSpeech2 [2], and AlphaGoZero [36], as our workloads as a representative of convolution neural networks,
workloads on the two baselines and on SAGAR. The distributed $4 \times 4$ system has much higher number of reads as compared to SAGAR and the monolithic baseline. In SAGAR this efficiency loss in reuse is mitigated by using bypassing links. As shown in Figure 11(b), across all layers in our workloads, SAGAR incurs SRAM reads close to that in the monolithic baseline. In the case of DeepSpeech2, SAGAR, owing to efficient mapping, incurs reads even fewer than that of the monolithic baseline. Similar trends are also reflected in other networks as well (Figure 11(g)). To further quantify the efficiency of SAGAR, we estimated the energy spent by the three configurations on the workloads by taking into account the cycle counts and the SRAM reads and scaling the counts by typical energy consumed per operation computed from RTL PnR flows. For all the workloads, the wire energies calculated using $100 \text{ fJ/bit-mm}$ at $14\text{nm}$ [7], come to be about $0.1\%$ (maximum being $0.11\%$ or $0.8\text{uJ}$ in AlphaGoZero), which is negligible. In Figure 11(d) we plot the energy consumed for the three workloads on the baselines and SAGAR. We observe that for workloads amenable to monolithic array (ie. FasterRCNN and DeepSpeech2), SAGAR’s energy consumption is almost identical to the monolithic baseline. The distributed baseline on the other hand consumes an order of magnitude higher energy for all the three workloads, while supporting the same mapping configurations as SAGAR. The difference in energies are a direct consequence of utilization. Since fine grained power or clock gating is impractical, the arrays with poor utilization consume same amount of power as the arrays with better utilization. However, these arrays take longer to complete resulting in higher energy consumption. For AlphaGoZero, which favours a distributed configuration, SAGAR consumes about $20\%$ of the energy consumed by the monolithic baseline, while almost one order of magnitude lower than that of the distributed baseline. Figure 11(d) also shows that SAGAR’s energy consumption for SRAM is close to that of consumed by the monolithic array for all the three workloads. The computation energy consumption in SAGAR equivalent to the better of the two baselines. The combined effect of improved latency and reuse is perhaps better represented by the energy-delay product (EDP) depicted by Figure 11(e). In this figure we plot the EDP for SAGAR and the two baselines normalized to the values corresponding to the monolithic configuration. We observe that SAGAR results in about $92\%$ to $80\%$ less EDP compared to the monolithic baseline. This further demonstrates the efficiency of our proposed architecture, resulting from
preserving reuse while simultaneously decreasing latency due to improved mapping.

B. Implementation evaluations

Methodology. We implemented SAGAR in RTL as a 32 × 32 array of 4 × 4 systolic-cells and ran ASIC flow till Place-and-Route (PnR) to obtain area and power. We used 28nm library for implementing the logic. We also implemented the SRAM buffers as a collection of 1024 1KB cells with the SAED32 education library from Synopsis, to quantify the power and area overheads, and then scaled down to 28nm equivalent by using Dennard’s scaling [9]. Figure 13(a) depicts the post PnR floorplan of SAGAR’s compute logic. Figure 13(b) lists the array configuration, area, and power consumption reported after PnR by synthesizing the RSA and memory at a operating frequency of 1 GHz. At 32.768 TOPs (with 1 MAC being two operation) at 1 GHz SAGAR takes 81.90 mm² of real estate while consuming 13.01 W of power. ADAPTNETX consumes 8.65% of area and 1.36% of power.

Baselines. We implement the baseline monolithic 128 × 128 systolic array and distributed 4 × 4 array in RTL. The distributed array is implemented using 1024 identical 4 × 4 traditional systolic arrays connected together by a mesh interconnect. We used the OpenSMART [21] tool to generate and synthesize the mesh topologies for these systems.

The total memory capacity of both the monolithic and the distributed configurations are kept the same at 3MB. As discussed in Section V-A the monolithic array has two input operand buffer of 1MB each and an output buffer also with the 1MB capacity. In our implementation, we opted for one bank per row or column of the array. This choice ensures that each incoming link to the array will have full bandwidth from SRAM provided that bank conflicts are negligible. Therefore each buffer in the monolithic baseline is constructed using 128, 8KB banks. For the distributed configuration, for each 4 × 4 array we end up with 1MB for each operand buffer. Using the same design approach as above, we end up with each buffer being constructed using 4 banks of 256 words each. In Table III we extend the same design principle for designing the memory for various other cell sizes and for SAGAR. In SAGAR, in addition to the links going directly from the SRAM to the edge MAC units of the array, we have to consider the bypass links as well. To get full bandwidth on these links we need to consider additional buffers. Extending the design described in Figure 5, each row and column of SAGAR has 31 bypass links and one link to the first MAC unit, we need 32 banks per row/column. Therefore each SRAM buffer is constructed with 1024, 1KB banks.

Area Analysis. In Figure 13(c) we depict the break down of area overheads for SRAM buffers, mesh NoC, and the compute array for various distributed configurations, the monolithic array, SAGAR and SIGMA [30]. We observe that the monolithic configuration is the most efficient in terms of area, where it is about 5× more compact than the distributed 4 × 4 array configuration. The breakdown suggests that the bloating in the distributed 4 × 4 configuration is caused predominantly by the Mesh NoC (contributing to 40.5%), followed by the SRAM buffers. SAGAR on the other hand takes about 8% more area than the monolithic array, while consuming about 3.2× lower area than the distributed 4 × 4 configuration. Considering both SAGAR and the distributed configuration provides same mapping flexibility, the proposed design is strictly more efficient.

Power Consumption. In Figure 13(d) we depict the post PnR power consumption for various array configuration. The Mesh NoC stands out as the major contributor, which naturally makes the 4 × 4 distributed configuration about 5.3× more expensive than the monolithic configuration, with the NoC contributing to about 78% of the power. Considering the power of the compute array alone, all the systolic-array based configurations appear to consume similar power. We also depict the trend in power consumed by SRAM banks across various systolic-array based configurations in Figure 13(f). Similar to the trends observed in area breakdown, the counter balancing affects of increasing the bank sizes and lowering of number of banks lead to similar powers across various distributed and monolithic configurations. RSA however consumes about 50% more power than the monolithic configuration, owing to the bypass links. However this extra cost results in achieving the same mapping flexibility of the 4 × 4 distributed configuration, which is about 3.5× more expensive.

Scalability Analysis. (i) Figure 13(g) we show the overhead of using smaller systolic-cell sizes in terms of area and power normalized to monolithic configuration. For specific use cases with relaxed requirements for flexibility larger sized systolic-cells can be used to improve the implementation costs. (ii) Figure 13(h) we depict the max frequency that can be met as a function of number of 4 × 4 systolic-cells that can be bypassed at 28 nm. Since we target 1GHz, we need to pipeline the bypass paths by inserting flops after 8 systolic-cells as we discuss in Section II-C.

C. Comparison with SIGMA

Implementation Comparison. We compare the area of SAGAR with the published area and power numbers of a state-of-the-art flexible accelerator SIGMA [30]. SIGMA allocates a significant portion of area for NoC, which together with SRAM comprise about 80% of the total area Figure 13(c). In SAGAR, simple bypass links are used to achieve the flexibility, which saves about 30% of the area in comparison. From Figure 13(d), we observe that NoC is SIGMA consumes about 1.8× more power than SAGAR, with NoC consuming 45% of total power.

Performance Comparisons. We use the analytical model used in the original paper 3 to estimate performance of SIGMA [30], which accounts for the time taken to stream, compute, and add partial sums as per the functionality described in their paper. In Figure 14(a) we plot the simulated runtimes for SAGAR, monolithic baseline, and SIGMA with equal number of MAC units (denoted as SIGMA_C) for our representative workloads and the ten layers reported in the SIGMA paper. SIGMA_C outperforms SAGAR in all workloads. This is due

3We thank the authors of SIGMA for their gracious support
in performance further widens with the increase in sparsity as shown in Figure 14(c).

As SIGMA implementation takes more area than SAGAR, we also compare against the area normalized configuration (2734 MACs) of SIGMA (denoted as SIGMA_A) in Figure 14 for fairness. In this case, SIGMA_A consumes about an order of magnitude more number of cycles for each workload as compared to compute normalized configuration, therefore rendering SAGAR as the best performer (Figure 14(b)). Even when considering workloads with sparse operands, SIGMA_A is able to surpass SAGAR only at operand sparsity values above 70% (see Figure 14(d)).

VI. RELATED WORKS

Flexible DNN Accelerator. Table I depicts the standing of various such accelerators in terms of native operation supported, mapping capability and flexibility. Designs like [18], [24], [37], [13], [10], [30], [32], [1], [12], [37] have limited flexibility in either reconfigurability or dataflow. The RECONFIGURABLE SYSTOLIC ARRAY enables both mapping flexibility and reconfigurability.

Dataflow and Accelerator Design Space Search. Contemporary tools [33], [20], [11], [28], [8] etc enable DSE by fast cost estimation or heuristics. SARA systems like SAGAR on the other hand obtain optimized configuration at runtime in one-shot using ADAPTNET.

ML assisted system configuration. Recent work [16], [17], [27] show using GAs for efficient search. RL and recommendation has been used for chip PnR [23], [25]. AutoTVM [4] use ML models for cost prediction to improve compilation time. It is worth noting that these approaches mostly enhance search for the optimal configuration, while ADAPTNET replaces search.
VII. CONCLUSIONS

This work shows that the mapping and configuration space of reconfigurable accelerator can be learnt using ML. We demonstrate this by developing a recommendation model called ADAPTNET which learns and predicts the optimal configurations for RSA with high accuracy. RSA is a flexible, scalable GEMM accelerator constructed using systolic-cells and pipelined bypass paths.

REFERENCES

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer cnn accelerators,” in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2016, pp. 1–12.
[2] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzano, Q. Cheng, G. Chen et al., “Deep speech 2: End-to-end speech recognition in english and mandarin,” in International Conference on Machine Learning, 2016, pp. 173–182.
[3] 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.
[4] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Cze, C. Guestrin, and A. Krishnamurthy, “Learning to optimize tensor programs,” in Advances in Neural Information Processing Systems, 2018, pp. 3389–3400.
[5] 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.
[6] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, “Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,” IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 9, no. 2, pp. 292–308, 2019.
[7] W. J. Dally, Y. Taurakhia, and S. Han, “Domain-specific hardware accelerators,” Communications of the ACM, vol. 63, no. 7, pp. 48–57, 2020.
[8] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shirivastava, “Dmarunner: Executing perfectly nested loops on dataflow accelerators,” ACM Transactions on Embedded Computing Systems (TECS), vol. 18, no. 5, pp. 1–27, 2019.
[9] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” IEEE Journal of Solid-State Circuits, vol. 9, no. 5, pp. 256–264, 1974.
[10] J. Fowers, K. Ovtcharov, M. Pampinich, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi et al., “A configurable cloud-scale dnn processor for real-time ai,” in 2019 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2018, pp. 1–14.
[11] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory,” in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 751–764.
[12] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, “Tangram: Optimized coarse-grained dataflow for scalable nn accelerators,” in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 807–820.
[13] S. Ghodrati, B. H. Ahn, J. K. Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim et al., “Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks,” in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020, pp. 681–697.
[14] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, “Mind mappings: enabling efficient algorithm-accelerator mapping space search,” in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 943–958.
[34] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “Scale-sim: Systolic cnn accelerator simulator,” arXiv preprint arXiv:1811.02883, 2018.

[35] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina et al., “Simba: Scaling deep-learning inference with multi-chip-module-based architecture,” in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 14–27.

[36] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton et al., “Mastering the game of go without human knowledge,” Nature, vol. 550, no. 7676, pp. 354–359, 2017.

[37] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2015, pp. 161–170.

[38] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, “mRNA: Enabling efficient mapping space exploration for a reconfiguration neural accelerator,” in 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2019, pp. 282–292.