A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim

Ananda Samajdar* Jan Moritz Joseph*† Yuhao Zhu‡ Paul Whatmough§ Matthew Mattina§ Tushar Krishna*

*Georgia Tech, Atlanta, GA, USA
†Otto-von-Guericke Univ., Magdeburg, Germany
‡Univ. of Rochester, Rochester, NY, USA
§ARM ML Research Lab, Boston, MA, USA
*anandsamajdar@gatech.edu

***Abstract*—The compute demand for deep learning workloads is well known and is a prime motivator for powerful parallel computing platforms such as GPUs or dedicated hardware accelerators. The massive inherent parallelism of these workloads enables us to extract more performance by simply provisioning more compute hardware for a given task. This strategy can be directly exploited to build higher-performing hardware for DNN workloads, by incorporating as many parallel compute units as possible in a single system. This strategy is referred to as *scaling up*. Alternatively, it's feasible to arrange multiple hardware systems to work on a single problem to exploit the given parallelism, or in other words, *scaling out*. As DNN based solutions become increasingly prevalent, so does the demand for computation, making the scaling choice (scale-up vs scale-out) critical.**

To study this design-space, this work makes two major contributions. (i) We describe a cycle-accurate simulator called SCALE-SIM for DNN inference on systolic arrays, which we use to model both scale-up and scale-out systems, modeling on-chip memory access, runtime, and DRAM bandwidth requirements for a given workload. (ii) We also present an analytical model to estimate the optimal scale-up vs scale-out ratio given hardware constraints (e.g, TOPS and DRAM bandwidth) for a given workload. We observe that judicious choice of scaling can lead to performance improvements as high as $50\times$ per layer, within the available DRAM bandwidth. This work demonstrates and analyzes the trade-off space for performance, DRAM bandwidth and energy, and identifies sweet spots for various workloads and hardware configurations.

I. INTRODUCTION

In recent years, the urgent compute demands stemming from Deep Neural Network (DNN) workloads have reinvigorated research into computer architecture, systems and high performance software design. The raw parallelism and reuse opportunities have spring-boarded devices like GPUs, which formerly were considered special purpose into ubiquity. This trend has also engendered a whole breed of hardware accelerators, which are aimed at squeezing out extremely high performance within commodity power and area budgets, owing to their custom design [28], [29], [32].

Nonetheless, within the field of deep learning, the hunger to consume more compute power seems insatiable. As the machine learning community develops deep learning based solutions for newer more complex problems, the DNN models become larger and more compute intensive. Furthermore, DNN-based methods are now being deployed to an ever increasing suite of applications, which means that the frequency of encountering a DNN-based workload is increasing as well [5], [23]. For computer architecture, this trend simply translates into the need to create more powerful and efficient hardware, or in other words to *scale* the performance of the hardware system.

The fundamental approaches to building efficient DNN accelerators have become fairly standard: since the majority of computation is some form of matrix-matrix multiplication[1], DNN accelerators typically employ a regular array of multiply-accumulate (MAC) units to compute these efficiently by leveraging data reuse within the array. The current differences seen across published accelerator microarchitectures today mainly lie in the memory hierarchy [12], [16]–[18], [20] and dataflow strategies [4] employed.

The optimal approach to scalability, however, remains an open question. Extracting higher performance essentially translates into allocating more parallel compute for a given workload. One way of achieving this is by creating a monolithic array with a large number of MAC units. The Google TPU [12] is an example of such a design. This approach is known as ***Scale-UP***. Alternatively, the effect of scaling can also be achieved by allocating multiple such units to collaboratively work on a given problem, or in other words by ***Scale-OUT***. Microsoft's Brainwave [7] is an instance of such a design. In fact, scale-out need not be across separate chips; NVIDIA's approach of a multitude of loosely coupled tensor cores [1] across SMs can also be viewed as an instance of scale-out. Both approaches involve a number of trade-offs. While a monolithic scale-up design provides the opportunity to exploit reuse, hence avoiding costly off-chip accesses, the operand reuse opportunities are of course finite, which ultimately limits utilization. On the other hand, a scale-out design may provides more mapping flexibility to maximize hardware utilization may additionally also be cheaper to design and (re-)configure.

In this paper, we analyze, identify, and quantify the components of this trade-off space to provide a systematic approach

SCALE-SIM was developed jointly at ARM Research, Boston and Georgia Tech. Download link: https://github.com/ARM-software/SCALE-Sim

[1]In this case matrix-matrix multiplication also encompasses the degenerate cases of matrix-vector and vector-vector products, where one or both operand matrices have dimension equal to one.

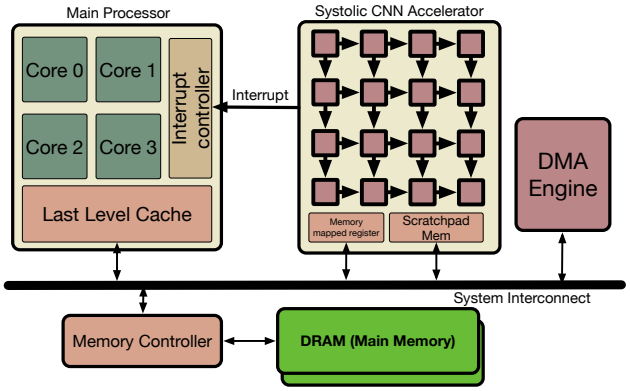


Fig. 1: Schematic showing the integration model of accelerator in a systems context

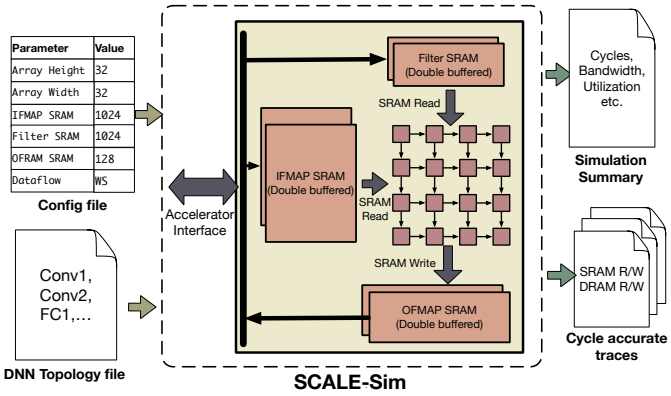


Fig. 2: Schematic depicting the inputs needed and the outputs generated by SCALE-SIM

for making scaling decisions. To enable this study, we develop a cycle accurate simulator for DNN accelerators called *SystoliC AcceLErator SIMulator* (SCALE-SIM) [24], which models compute performance, on-chip and off-chip memory accesses, and interface bandwidth information for a given neural network. SCALE-SIM implements two elements: (i) a compute unit based around a systolic array parameterized by size and aspect ratio, and (ii) a simple accelerator memory system with three double buffered SRAM memories of user specified sizes, which buffers the matrices for two operands and one result. The inputs to the tool are the layer dimensions of a given neural network workload, and the hardware architecture parameters. SCALE-SIM can model both scale-up (one partition) and scale-out (multiple partition) instances.

To allow for fast design space exploration and rapid identification of design insights, we augment the simulator with an analytical model that captures the first-order execution time of a single systolic array. Unlike SCALE-SIM, the analytical model does not consider cycle by cycle accesses and bandwidth demands due to limited memory sizes. Instead, it captures the first-order performance, and thus helps prune the search space. As we will describe in Section III, we use this model to determine the most performant configuration for both monolithic (scale-up) and partitioned (scale-out) systems for a given workload.

Using SCALE-SIM augmented with analytical models, we systematically explore the design space of DNN accelerators, focusing on understanding the trade-off of scale-up versus scaling-out configurations in Section IV. We find that the fundamental trade-off is between performance and DRAM bandwidth demands. Finally, we propose a heuristic-driven approach that efficiently identifies the optimal scaling strategy, along with the design configuration within a particular scaling strategy, for a given set of workloads. In summary the following are the main contributions of this paper:

1) We develop SCALE-SIM, a cycle accurate, configurable systolic array based DNN accelerator simulator;
2) We develop an analytical model for compute the runtime of DNNs on a systolic array, and using this to determine the optimal size, aspect ratio and number of partitions for achieving the best performance for a given workload;
3) We present an in-depth study of the trade-off space pertaining to scaling decisions in DNN acceleration.

II. SCALE-SIM: SYSTOLIC ACCELERATOR SIMULATOR

SCALE-SIM is a cycle-accurate behavioural simulator that provides a publicly available open-source modeling infrastructure for array-based DNN accelerators. SCALE-SIM enables designers to quickly iterate over and validate their upcoming designs with respect to the various optimization goals for their respective implementation points. In this section, we first provide some background on systolic arrays and second, we describe our modeling methodology.

A. Background: Systolic Arrays and Dataflows

Systolic arrays are a class of simple, elegant and energy-efficient architectures for accelerating general matrix multiplication (GEMM) operations in hardware. They appear in many academic and commercial DNN accelerator designs [2], [12], [14]. An overview of system integration is shown in Figure 1

Compute. The compute microarchitecture comprises several Multiply-and-Accumulate (MAC) units (also known as Processing Elements, or PEs), connected in a tightly coupled two dimensional mesh. Data is fed from the edges from SRAMs, which then propagates to the elements within the same row (column) via unidirectional neighbour-to-neighbour links. Each MAC unit stores the incoming data in the current cycle in an internal register and then forwards the same data to the outgoing link in the next cycle. This store and forward behavior results in significant savings in SRAM read bandwidth and can very effectively exploit reuse opportunities provided by convolution/GEMM operations, making it a popular choice for accelerator design. Note that this data movement and operand reuse is achieved: (1) without generating or communicating any address data, and (2) only using hard-wired local register-to-register inter-PE links, without any interconnect logic or global wires. For these two reasons, the systolic array is extremely energy and area efficient.

Memory. Systolic Arrays are typically fed by local linearly-addressed SRAMs on the two edges of the array, with outputs collected along a third edge. These local SRAMs are often double buffered and are backed by the next level of the memory hierarchy.

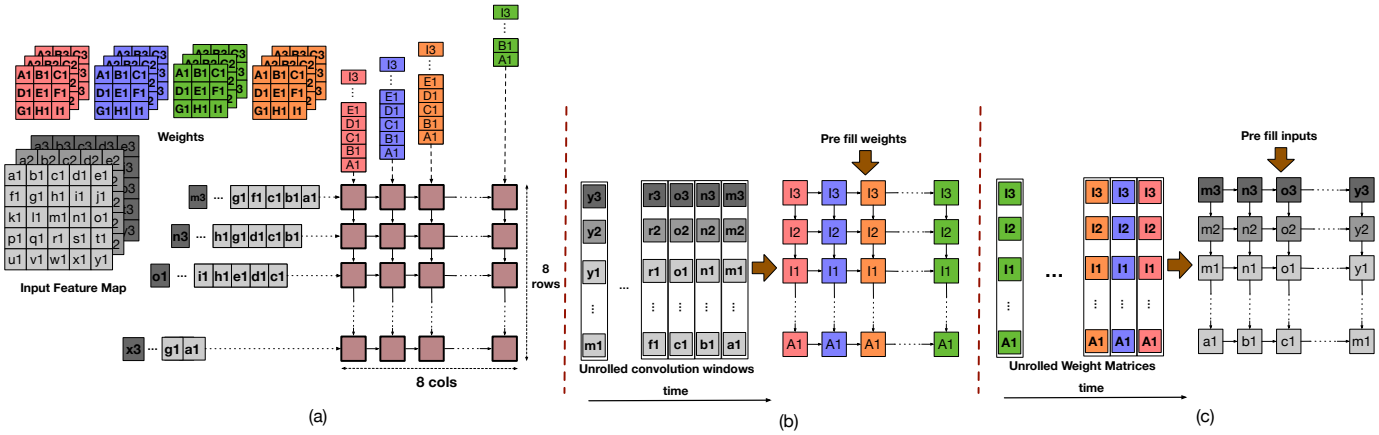


Fig. 3: Schematic showing the mapping in various dataflows (a) Output stationary; (b) Weight stationary; (c) Input stationary

Data Reuse. A typical convolution can be viewed as a small filter kernel being slid over a given input matrix, with each overlap generating one output pixel. When the convolution operation is formulated as successive dot-product operations, three reuse patterns are immediately evident:

> Each convolution window uses the same filter matrix, to generate pixels corresponding to a given output channel. The adjacent convolution windows share portions of the input matrix if the stride is smaller than window dimension.
>
> To generate a output pixel in different output channels, different filter matrices use the same convolution window.

These reuses can be exploited via the dataflow or mapping of the DNN over the array.

Dataflow. There are three distinct strategies of mapping compute or *dataflows* onto the systolic array named *Output Stationary (OS)*, *Weight Stationary (WS)*, and *Input Stationary (IS)* [4] as shown in Figure 3. 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, we only consider true systolic dataflows that only use local communication.

The OS dataflow depicted in Figure 3(a), therefore refers to the mapping where each MAC units is responsible for all the computations required for a OFMAP pixel. All the required operands are fed from the edges of the array, which are distributed to the MAC processing elements (PE) using internal links to the arrays. The partial sums are generated and reduced within each MAC unit. Once all the MAC units in the array complete the generation of output pixels assigned to itself, the peer to peer links are used to transfer the data out of the array. No computation takes place in the array during this movement. An alternative high performance implementation using a separate data plane to move generated output is also possible, however, it is costly to implement.

The WS dataflow on the other hand uses a different strategy as shown in Figure 3(b). The elements of the filter matrix are pre-filled and stored into each PE prior to the start of computation, such that all the elements of a given filter are allocated along a column. The elements of the IFMAP matrix are then streamed in through the left edge of the array, and each PE generates one partial sum every cycle. The generated partial sums are then reduced across the rows, along each column in parallel to generated one OFMAP pixel (or reduced sum) per column.

The IS dataflow is similar to WS, with the difference being in the order of mapping. Instead of pre-filling the array with elements of the filter matrix, elements of the IFMAP matrix are stored in each PE, such that each column has the IFMAP elements needed to generate a given OFMAP pixel. Figure 3(c) depicts the mapping. We describe these dataflows in more detail in Section III-B.

B. System Integration

We consider the typical offload model of accelerator integration in SCALE-SIM. We attach the DNN accelerator to the system interconnect, using a slave interface on the accelerator, as illustrated in Figure 1. The CPU is the bus master which interacts with the accelerator by writing task descriptors to memory-mapped registers inside the accelerator. When a task is offloaded to the accelerator, the CPU master can context switch to progress other jobs, while the accelerator wakes up and starts computing, independently generating its memory requests and side channel signals. When the computation has finished, the accelerator notifies the CPU, which accesses the results from the accelerator internal memory.

Thus, the cost on the system performance for integrating an accelerator is the extra accesses on the system bus, which could be modelled as interface bandwidth requirement. SCALE-SIM allows for modeling the main memory behavior by generating accurate read and write bandwidths at the interface, which can then be fed into a DRAM simulator e.g., DRAM-Sim2 [22].

C. Implementation

Internally, SCALE-SIM takes an inside-out implementation approach. Specifically, the simulator assumes that the accelerator is always compute bound and the PEs are always used to the maximum possible utilization - as dictated by the dataflow

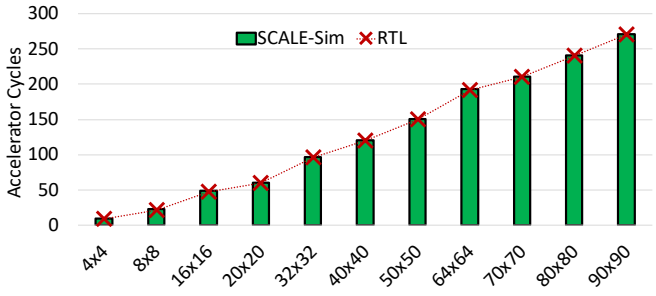


Fig. 4: Figure depicting the cycles obtained by RTL implementation and SCALE-Sim simulation for varying array sizes under full utilization

in use. With this implementation model, the simulation in SCALE-SIM takes place in following steps.

SCALE-SIM generates cycle accurate read addresses for elements required to be fed on the top and left edges of the array *such that the PE array never stalls*. These addresses are effectively the SRAM read traffic for filter and input matrices, as dictated by the dataflow. Given the reduction takes a deterministic number of cycles after the data has been fed in, SCALE-SIM generates an output trace for the output matrix, which essentially constitutes the SRAM write traffic.

SCALE-SIM parses the generated traffic traces, to determine total runtime for compute and data transfer to and from SRAM. The data transfer time is essentially the cycle count of the last output trace entry. The SRAM trace also depicts the number of rows and columns that have valid mapping in each cycle. This information couples with the dataflow is used to determine the utilization of the array, every cycle.

In SCALE-SIM the elements of both the input operand matrices, and the generated elements of the output matrix is serviced by dedicated SRAM buffers backed via a double buffered mechanism, as shown in Figure 2. As the sizes of these buffers are known from user inputs, SCALE-SIM parses the SRAM traces and determines the time available to fill these buffers such that no SRAM request is a miss. Using this interfaces SCALE-SIM generates a series of prefetch requests to SRAM which we call the DRAM trace.

The DRAM traces are the used to estimate the interface bandwidth requirements for the given workload and the provided architecture configuration.

The trace data generated at the SRAM and the interface level is further parsed to determine the total on-chip and off-chip requests, compute efficiency, and other high level metrics.

D. Validation of the tool

We validated SCALE-SIM against an RTL implementation of a systolic array. Figure 4 depicts the cycles obtained when matrix multiplications are performed on varying arrays sizes (X-axis) under full utilization with OS dataflow, from RTL implementation and SCALE-SIM simulations. As depicted by the figure the cycle counts obtained by both the methods are in good agreement.

TABLE I: SCALE-SIM config description

Parameter	Description
ArrayHeight	Number of rows of the MAC systolic array
ArrayWidth	Number of columns of the MAC systolic array
IfmapSRAMSz	Size of the working set SRAM for IFMAP in KBytes
FilterSRAMSz	Size of the working set SRAM for filters in KBytes
OfmapSRAMSz	Size of the working set SRAM for OFMAP in KBytes
IfmapOffset	Offset to the generated addresses for IFMAP px
FilterOffset	Offset to the generated addresses for filter px
OfmapOffset	Offset to the generated addresses for OFMAP px
DataFlow	Dataflow for this run. Legal values are 'os','ws', and 'is'
Topology	Path to the topology file

TABLE II: SCALE-SIM Topology file description

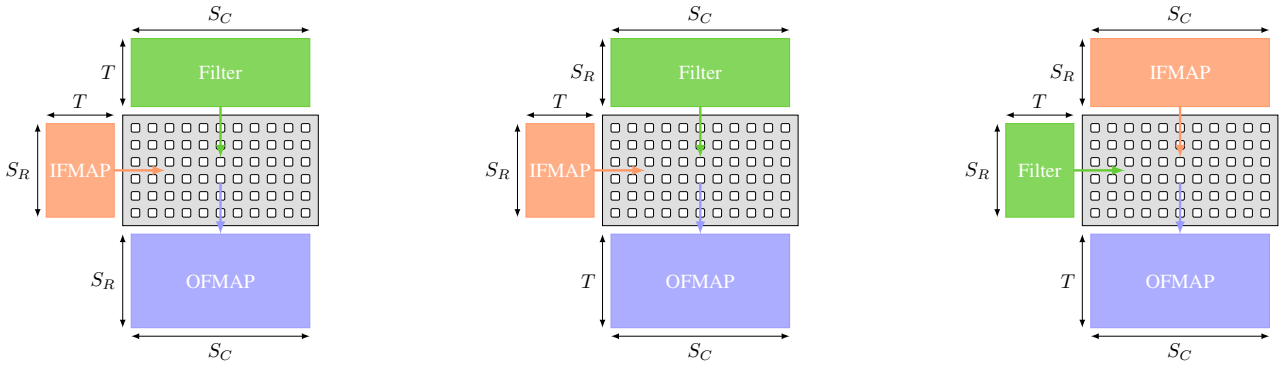
Parameter	Description
Layer Name	User defined tag
IFMAP Height	Dimension of IFMAP matrix
IFMAP Width	Dimension of IFMAP matrix
Filter Height	Dimension of one Filter matrix
Filter Width	Dimension of one Filter matrix
Channels	Number of Input channels
Num Filter	Number of Filter matrices. This is also the number of OFMAP channels
Strides	Strides in convolution

E. User Interface

Figure 2 depicts the inputs files used by the simulator, the outputs that are generated. SCALE-SIM takes two files as input from the user: one is a hardware configuration, and the other is a neural network topology for the workload. The configuration file contains the user specification for architectural parameters, like the array size, the memory size, and the path to the topology file. Table I depicts the complete list of parameters, which are mostly self-explainatory. For layers such as fully-connected (i.e. matrix-vector), the input parameters correspond to convolutions where the size of the filters are same as that of the IFMAP.

The topology file contains the layer topology dimensions for each of the layers in the given neural network workload. This is a comma-separated value (CSV) file, with each row listing all the required hyper-parameters for a given layer – Table II gives the complete list of all the entries in a given row. SCALE-SIM parses the topology file one line at a time and simulates the execution of the layer. This is a natural approach for traditional neural networks which are primarily composed of a single path. However, modern DNNs often contain "cells" that are composed of multiple convolution layers in parallel [10]. SCALE-SIM serializes the execution of such layers in the same order in which they are listed in the topology file.

SCALE-SIM generates two types of outputs. First is the cycle accurate traces for SRAM and DRAM reads and writes. The traces are also CSV files, which list the cycle and the addresses of data transferred in a given cycle. The other type of output files are reports with aggregated metrics obtained by parsing information from the traces. These include cycle counts, utilization, bandwidth requirements, total data transfers etc. The trace-based methodology is very easy to debug and highly-extensible to new analyses and architectures.



(a) Output stationary dataflow. (b) Weight stationary dataflow. (c) Input stationary dataflow.

Fig. 5: Data Flow Mapping.

TABLE III: Spatio-Temporal Allocation of DNN Dimensions

	Spatial Rows (S_R)	**Spatial Columns** (S_C)	**Temporal** (T)
Output Stationary	N_{ofmap}	N_{filter}	W_{conv}
Weight Stationary	W_{conv}	N_{filter}	N_{ofmap}
Input Stationary	W_{conv}	N_{ofmap}	N_{filter}

N_{filter} : Number of convolution filters
N_{ofmap}: Number of OFMAP pixels generated by filter
W_{conv} : Number of partial sums generated per output pixels

III. Analytical Model for Runtime

In SCALE-Sim, all the simulated metrics including runtime are determined at the end of a round of simulation. However running simulation for all possible data points in a large search space is expensive and sometimes unnecessary. In this section we describe an effective analytical model for runtime, which accounts for the data movement patterns simulated by SCALE-Sim. Please note however, the analytical model does not model the memory accesses and bandwidth demand arising due to limited memory which is captured by SCALE-Sim. We use this model to estimate costs and prune the search space for the subsequent scalability study described in Section IV.

A. Mapping across Space and Time

In dense DNN computations, running different types of layers generalize to matrix-matrix multiplications of different sizes. For systolic arrays, we consider the operand matrices of dimensions $S_R \quad T$ and $T \quad S_C$ respectively, where S_R and S_C are the spatial dimensions along which computation is mapped, and T is the corresponding temporal dimension. These matrices are obtained by projecting the original operand matrices into the available spatio-temporal dimensions. For example, for multiplying matrices of size $M \quad K$ and $K \quad N$, the dimension M is mapped to S_R, dimension N is mapped to S_C and the dimension K to T.

Figure 5 illustrates the mapping of a 2D convolution onto the three dataflows. Figure 5a shows the mapping corresponding to output stationary (OS) dataflow. The first operand matrix, with size $S_R \quad T$, is a rearranged input feature map (IFMAP) matrix. Each row consists of elements corresponding to one convolution window, while the number of rows is the number of OFMAP pixels generated per filter. The second operand matrix contains unrolled filter elements, with each filter unrolled along each column, resulting in a $T \quad S_C$ matrix.

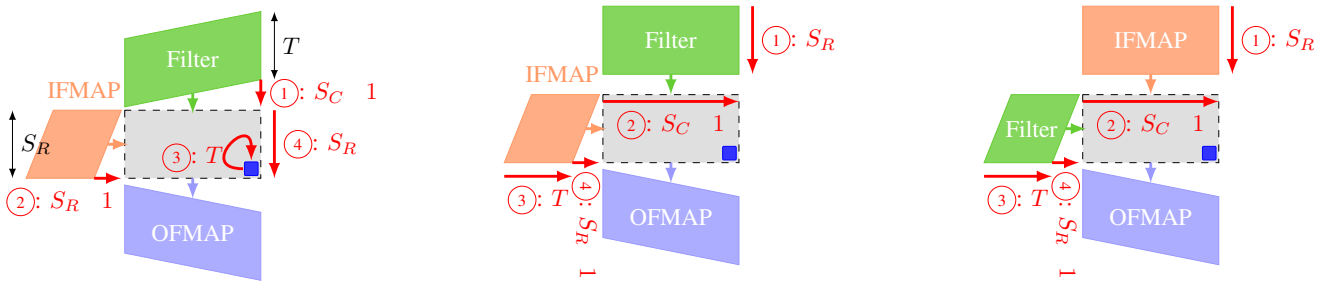
Figure 5b and Figure 5c depict the mapping for other two dataflows; *Weight Stationary (WS)* and *Input Stationary (IS)*. For WS, the number of convolution windows maps to S_R, while S_C is equal to the number of filters. As seen in Section II the partial sums for each OFMAP pixel are generated every subsequent cycle making the mapping along the temporal dimension T equal to the number of OFMAP pixels generated. In the IS dataflow however, the order and direction of feeding the IFMAP matrix and the filter matrices are interchanged. This implies that the mapping along the S_R and S_C dimensions for this dataflow is the same size as the convolution window and number of OFMAP pixels generated per filters respectively. While the temporal dimension T maps the number of filters. Table III summarizes these dimensions.

B. Runtime for Scale-Up

With the above abstraction of mapping in place, it is feasible to model the runtime for various dataflows, under the assumption of either a restricted or unrestricted number of compute elements. In our discussions we will only use multiply-and-accumulate (MAC) units as the compute elements within the systolic array.

1) Runtime with unlimited MAC units: Given an unlimited amount of MAC units, the fastest execution for any dataflow is achieved using the maximal array size of $S_R \quad S_C$. However, note that even though all the multiplication operations are done in one cycle, the runtime needs to account for both the store and forward nature of the array, and the existence of the temporal dimension T (> 0).

Figure 6 shows the steps followed for moving data in the three dataflows introduced in Section II. Figure 6a depicts the steps when implementing the OS dataflow. As mentioned before the IFMAP matrix is fed from the left while the filter elements are pushed in from the top edge. To account for the store and forward nature of the arrays and match the data arrival time at all the PEs, the data distribution is skewed; the PE at the top left corner of the array receives both the operands at the first cycle, the PEs in the next column and next row get their operands in the next cycles, their neighbours



(a) Output stationary runtime. (b) Weight stationary runtime. (c) Input stationary runtime.

Fig. 6: Schematic depicting steps to model runtime for dataflows in systolic array.

in the cycle after that and so on. The PE at the bottom right corner of the array (marked in blue), is the last to receive the operand data. It is easy to see that the cycle at which the first operands arrive at this PE is $S_R + S_C$ 2 (adding steps (1), (2) and (3)). In this dataflow, each PE receives two operands per cycle and generates one OFMAP pixel value by in-place accumulation. It takes T cycles to generate on output, which is equal to the number of elements in a convolution window. The generated outputs are taken out from the bottom edge of the array. While it is possible to take out the output along other edges as well, using the bottom edge is the fastest alternative. The time required to completely drain the array of the generated output is S_R cycles after the PE at the right most corner has finished computation (step (4)). Therefore, the total time taken for entire computation is,

$$\tau_{scaleup\_min} = 2S_R + S_C + T \quad 2 \tag{1}$$

In Figure 6b we perform the same analysis for WS dataflow. Here, the filter matrix is fed into the array from the top and is kept alive untill the computations involving these operands are complete. Skewing is not needed as no computation is taking place while the filters are being fed. This takes S_R cycles (step (1)). Once the filter elements are in place, the elements of the IFMAP matrix are fed from the left edge of the array. Each PE reads the IFMAP operand, multiplies it with the stored weight and forwards the partial sum to the PE in the neighbouring row for reduction. The first data arrives at the last row after S_C 1 cycles (step (2)). The IFMAP matrix is fed in one column at a time, therefore every column in the systolic array receives T operands, one each cycle, corresponding to the number of columns in the IFMAP matrix (step (3)). Furthermore, for all the partial sums generated reduction occurs across the rows, for each column. After the top row receives and operand from the IFMAP, it takes S_R 1 cycles to reduce (step (4)). Therefore the array is drained out of all partial sums, after reduction happens in the rightmost column. The total runtime therefore is,

$$\tau_{scaleup\_min} = 2S_R + S_C + T \quad 2$$

Using similar analysis and Figure 5c, we can show that the above expression holds true for the IS dataflow as well. *Thus Equation 1 captures the runtime for all the dataflows in a systolic array when the number of MAC units is infinitely large*

2) Runtime with limited MAC units: Having a large enough systolic array which can map all the compute at once is often not practically feasible. Due to the large amount of computation compared to hardware compute units, it is necessary to tile the workload into chunks. We term this practice as *folding* where each of these chunks are called a *fold*[2]. Folds can be generated by slicing the compute along the S_R and S_C dimensions. When using a R C array, the number of fold along rows (F_R) and columns (F_C) are determined as follows.

$$F_R = \lceil S_R/R \rceil, \; F_C = \lceil S_C/C \rceil \tag{2}$$

Figure 7 illustrates this.

Analysis similar to Section III-B1 can be used to express the time taken in each of these folds as is given by the following equation, for all dataflows.

$$\tau_F = 2R + C + T \quad 2 \tag{3}$$

Where R and C are the rows and columns of the systolic array and T is the temporal dimensions. The total runtime can therefore be expressed from Equation 2 and Equation 3 as following.

$$\tau_{scaleup} = (2R + C + T \quad 2)\lceil S_R/R \rceil \lceil S_C/C \rceil \tag{4}$$

The above equation provides us with the insights on the factors affecting runtime. For a given workload and array configuration, choice of dataflow assigns the values for S_R, S_C and T respectively, which could be selected to minimize τ. On the other hand if the workload and dataflow is fixed, for a given number of MAC units, the optimal values of R and C could be determined to reduce the runtime as well.

Equation 4 can be used to determine the optimal configuration for a given matrix by implementing search over the possible R and C values. For workloads with multiple matrix operations, this model can be used as a cost model as depicted later in Section IV-B.

C. Optimal Partitioning for Scale-Out

In our previous analysis we have only considered a single array to study the affect of micro-architectural and design parameters on runtime. Instead of creating a single monolithic architecture with multiple PEs (i.e., scale-up), an alternative

[2]This is often also known as tiling

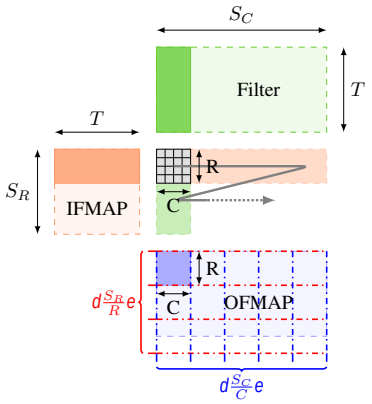


Fig. 7: Scale Up

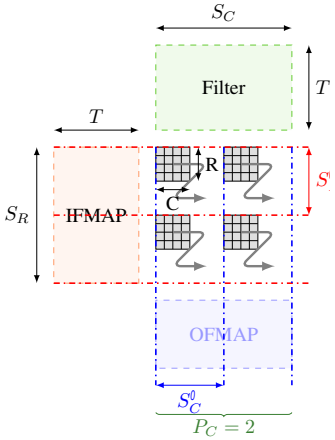


Fig. 8: Scale Out

design choice is to employ multiple units of systolic arrays, each responsible for one partition of the output feature map, to increase the available parallelism (i.e., scale-out) In this section we will model the runtime of such systems.

The scaled out configuration introduces another set of parameters, as shown in Figure 8. Unlike in scale-up where all the MAC units are arranged in a R C array, in scaled-out configuration, the MAC PEs are grouped into P_R P_C systolic arrays, each with a PE array of R C.

Using this approach for a given number of partitions $P = P_R$ P_C, the effective workload mapped for computation over each partition can be determined by,

$$S'_R = dS_R/P_Re,\ S'_C = dS_C/P_Ce \tag{5}$$

Within each array, we can use Equation 4 to decide the optimal aspect ratio (R C) for running the partitioned workload. Since the individual partitions execute in parallel, the total runtime of the scaled-out system is simply the runtime of the slowest cluster which can be determined by Equation 4 and Equation 5

$$\tau_{scaleout} = (2R + C + T \quad 2)dS'_R/RedS'_C/Ce \tag{6}$$

IV. Analysis of Scaling

The primary aim of scaling a hardware accelerator, is to improve the runtime of a given workload. Since there are many ways of scaling a system, the first natural question to ask is whether any one of the methods proves beneficial over the others. To answer this question, we computed runtime using the analytical model described in Section III, when using different configurations of monolithic vs scaled out arrays, given the same budget for MAC units. For workloads in our experiments, we used the convolution layers in Resnet50 CNN [10] and a few representative layers from widely used contemporary natural language processing models: GNMT [30], DeepSpeech2 [3], Transformer [26], and neural collaborative filtering [11]. The matrix dimensions corresponding to these workloads are detailed in Table IV

Search Space for Scale-up and Scale-out. Figure 9(a) provides the glimpse of the search space associated with the problem at hand. Each marker in the figure depicts a design

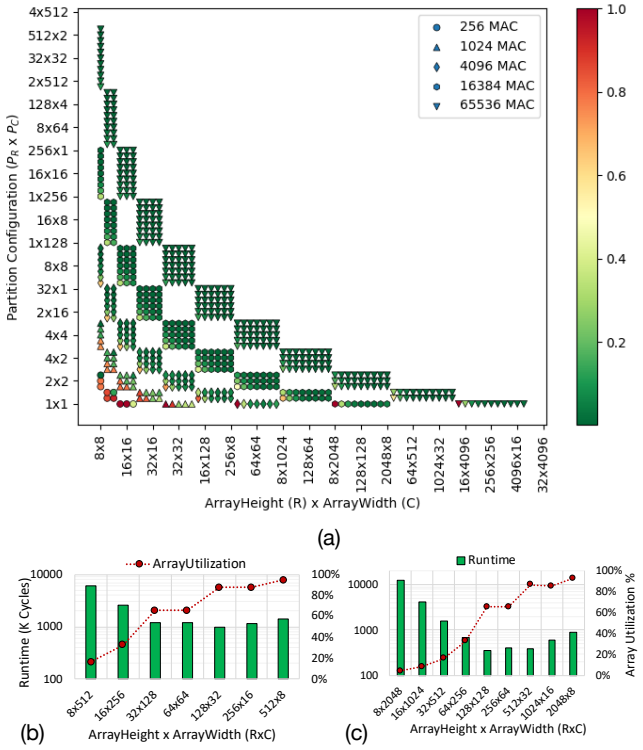


Fig. 9: (a) The search space of all possible scale-up (monolithic) and scale-out (partitioned) configurations, with different array sizes; the color represents runtime for TF0 layer of the Transformer model, normalized to max runtime across configurations for a given array size. The variation in runtime and array utilization for all scaled-up configurations when running TF0 layer for (b) 2^{14} MACs, (c) 2^{16} MACs.

TABLE IV: Matrix dimensions of our language model workloads. mapped to S_R, S_C, and T

Name	S_R	T	S_C
GNMT0	128	4096	2048
GNMT1	320	4096	3072
GNMT2	1632	1024	36548
GNMT3	2048	32	4096
DB0	1024	50000	16
DB1	35	2560	4096
TF0	31999	84	1024
TF1	84	4096	1024
NCF0	2048	128	1
NCF1	256	2048	256

point for corresponding to five different compute capabilities denoted by number of MAC units. On the x axis we have all possible dimensions for a systolic array with these mac units. The y axis represents the partitioned configurations when scaling out. We limit the smallest systolic dimensions to 8x8 to ensure we have a reasonable size arrays per partition when scaling out. The color of each point denotes the normalized stall free run time when TF0 is run using OS dataflow. Run times are normalized to the highest runtime among all the configurations for a fixed number of MAC units.

Effect of Aspect Ratio on Scale-up Array. From this chart we can get a first order estimate of runtime variation between partitioned and monolithic configurations. We observe that the highers runtimes are usually located near the points corresponding to y value of 1 1, which represent the mono-

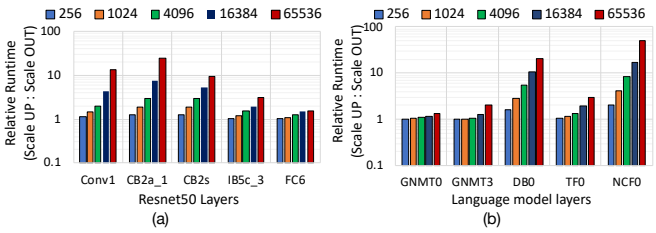


Fig. 10: Ratio of no stall runtimes obtained in best scaled-up array configuration vs best scaled-out (partitioned) configuration for a few layers in (a) Resnet50 and (b) Language models, for different MAC units

lithic configurations. Figure 9(b-c) depicts the various aspect ratio (Row:Column) configurations for monolithic arrays with 4096 and 16384 MAC units respectively. The first observation is that, the difference in runtime for optimum configuration and others can vary by several orders of magnitude even when the workload is the same, depending on the size of the array. In fact, with larger arrays this difference is exacerbated. Second, the aspect ratio of the optimal configuration is not the same at different performance points, necessitating the need to have a framework to examine various configurations. When considering the array utilization, another interesting trend arises. For configurations with low array utilization, the runtime of the layer is high, which is expected. Also, runtime generally drops with array utilization. Interestingly, when the array dimensions become significantly rectangular, the effect of utilization is less pronounced. In these configurations even though a high utilization is achieved, the improvement in runtime is minimal. This is due to the fact that the time to fill in and take out the data starts dominating, as captured in Equation 3.

Comparison of Best Runtime. Moving to the points up along the y-axis in Figure 9(a) show almost monotonic improvement in performance, depicting that partitioning is always beneficial. To further investigate this trend in Figure 10 we plot the stall free runtimes corresponding to the fastest scaled out (monolithic) configuration normalized to the lowest runtime achieved among all the scaled-out (partitioned) configurations using equal MAC units. Figure 10(a) plots the rations for first and last five convolution and fully connected layers of Resnet50 CNN for different number of MAC units. It can be observed that monolithic configurations are sometimes significantly slower (25x for CB2a_1 layer) that partitioned configurations, and never faster that the corresponding partitioned configuration. Moreover, for a given layer, the relative slowdown tends to amplify when the hardware is scaled. This trend is also replicated in language models, which predominantly use fully connected layers as seen in Figure 10(b). Here for 65536 MAC units the best monolithic configuration is 50x slower than the best partitioned configurations.

Note that since the runtimes involved in the above charts are stall free, the memory is not involved in slowdown. Therefore, the root cause of this slowdown can be understood by a closer look into the analytical model. First we should remember that in both monolithic and partitioned configurations the amount of serial computation is equal assuming all the MACs are utilized, or in other words the number of folds are equal. However from Equation 4 we can see that the runtime per fold is directly proportional to the array dimensions. Which explains the trend that the partitioned configurations are always faster. Furthermore, the difference in runtime per layer is amplified if the number of folds are high, even when both the arrays are fully utilized and the difference comes from data loading and unloading times. Also, utilizing the entire array in a monolithic configuration, howsoever flexible, is often not possible, as we can notice in Figure 9(b-c), which limits the amount of available compute resources and thus, contributes further to the relative slowdown.

A. *Cost of scaling out*

Observations from the experiments in the previous section seem to suggest that scaling out is the best strategy to achieve the optimal runtime. However this choice involves paying additional costs as we discuss below.

The immediate cost of a partitioned design is the loss of spatial reuse. In a big systolic array any element read from the memory is used by processing elements along a row or column by forwarding it on the internal links of the array. Dividing up the array into smaller parts reduces the number of rows, or columns, or both, resulting in drastic reduction of this reuse opportunity. This is then reflected in terms of number of SRAM reads, data replication, and the input bandwidth (BW) demand from the DRAM. The loss of reuse within the array over short wires also leads to longer traversals over an on-chip/off-chip network (depending on the location of the partitions) to distribute data to the different partitions and collecting outputs - which in turn can affect overall energy.

Runtime vs. DRAM BW Requirement. In Figure 11 we plot the DRAM BW requirement and runtime for layer CBa_3 in Resnet-50 and layer TF0 in Transformer, as a function of number of partitions, for given number of MAC units. For all the three cases a total of 512KB of SRAM is allocated for IFMAP buffer, 512KB for Filter buffer, and 256 KB for OFMAP buffer. This memory is evenly distributed among the partitions in case of scaling out. The BW numbers are obtained from our cycle accurate simulator when running the output stationary dataflow. As the number of partitions increase, the runtime goes down, however, BW requirements also rise due to loss of reuse originally provisioned by the internal wires, and increased replication of the data among the partitions, bringing down the effective memory capacity. The sweet spot lies at the intersection of runtime and bandwidth curves. When scaling to higher number of MAC units, it is interesting to note that the BW requirement is often higher than traditional DRAM BW. For instance, for both Resnet and Transformer layers with 2^{18} MAC units, about 10 KB/cycle of DRAM bandwidth is needed for stall free operation at the sweet spot.

Energy Consumption. In Figure 12 we study the effect of scaling out on energy. Figure 12(a) depicts the energy consumption to run layer CBa_3 of Resnet50 as the number of partitions are increased for various MAC unit (barring the

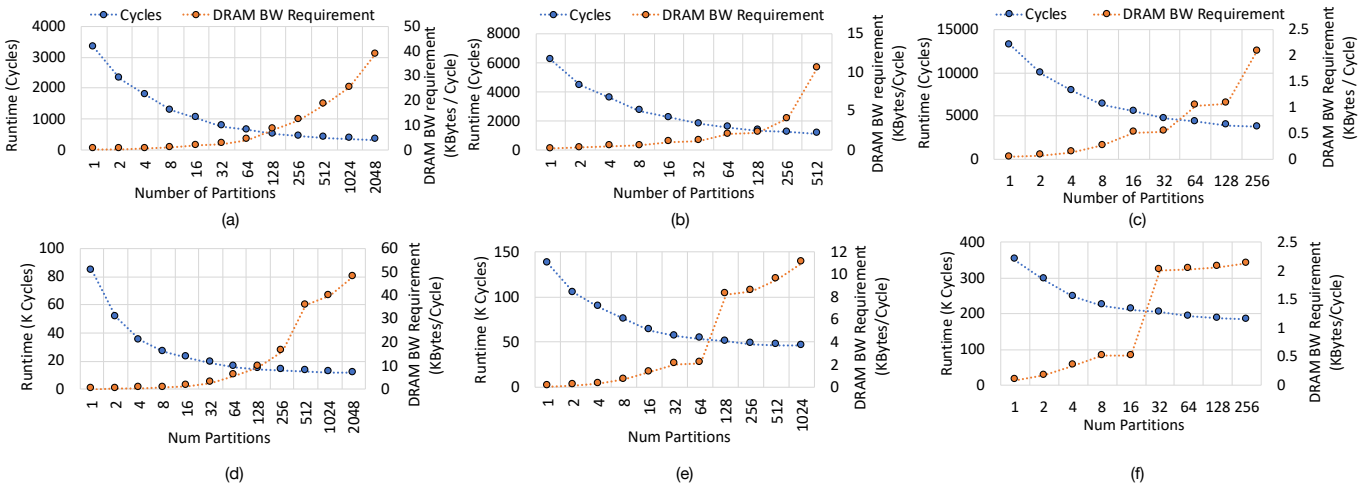


Fig. 11: Trends for best possible stall free runtime and DRAM bandwidth requirements when the number of partitions are increased from monolithic array in CBa_3 layer in Resnet50 for (a) 2^{18} MAC units, (b) 2^{16} MAC units, and (c) 2^{14} MAC units; and TF0 layer in Transformer for (d) 2^{18} MAC units, (e) 2^{16} MAC units, and (f) 2^{14} MAC units

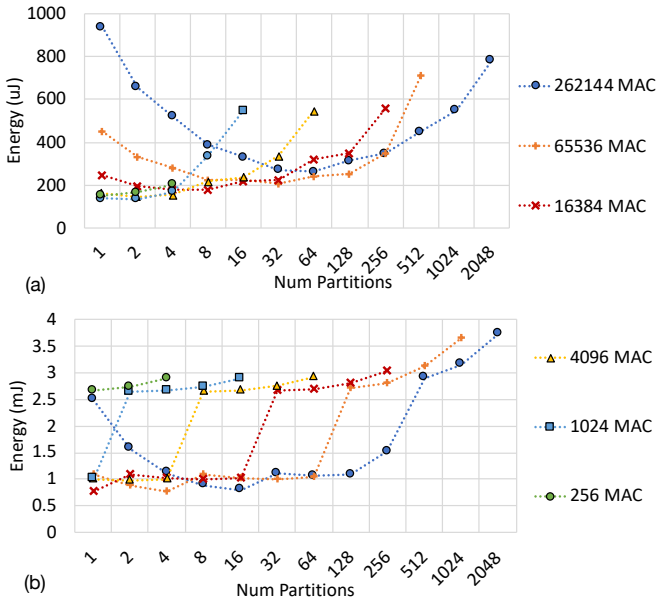


Fig. 12: Energy consumption in running (a) Layer CBa_3 from Resnet50 and (b) layer TB0 from Transformer, when scaling-up and scaling out with different MAC units

energy consumption of interconnection network). Figure 12(b) captures the same information for Layer TF0 for Transformer. For a given workload and hardware configuration, the energy consumption directly depends on the cycles MAC units have been active and the number of accesses to SRAM and DRAM. The counteracting effects of these factors can be observed in Figure 11, therefore lays down an interesting tradeoff space. As the figure depicts, for lower number of MAC units (256, 1024 and 4096), the configuration with minimum energy is the monolithic configuration. However with increase in number of MAC units, the point of minimum energy moves towards the right of the chart, favouring more number of partitions. On other words the energy saved in by stealing runtime from powering the massive compute array is more significant than the extra energy spent by the loss of reuse. Furthermore, the bulkier the array, more the savings in compute to counteract the losses in reuse, which explains the observed trend.

To summarize the data indicates scaling out is beneficial for performance and with larger MAC units is more energy efficient that scaling up. However the cost paid is the extra bandwidth requirement to keep compute units fed, which even at sweet spots are significantly higher than the best scaled-up configuration for large MAC units.

B. Optimizing for multiple workloads

Any hardware accelerator should be performant for different workloads. To find such a globally optimized hardware accelerator, a global cost function must be minimized. However, as Figure 9(a) depicts even for a single workload as the global cost function is large and discontinuous. Optimally searching such a space for finding the global minima is out of the scope of this paper. Instead we propose a method to find reasonable pareto-optimal points for a given set of workloads

Considering the runtime as cost, our analytical model from Sec. III-B and III-C or SCALE-SIM yields a runtime-optimal configuration, $a_k = (S_C^0, S_R^0, R, C)$, for each individual layer (i.e. workload $w_l = (S_C, S_R, T)$). We then search among these candidates for the globally optimized one, A. In case of runtime, the total runtime is additive and thus it is calculated by summing the runtimes T_r of all workloads w_l for each candidate a_k:

$$A = argmin_{a_k} \sum_{w_l} T_r(w_l, a_k)$$

As the number of candidates is limited, exhaustive search is feasible to find the optima.

In Figure 13 we plot the costs (runtime) of the various candidate configurations normalized to the cost of the pareto-optimal configuration obtained by the method mentioned above, for layers in Resnet50 and the language models mentioned in Table IV. In Figure 14 the normalized costs for all

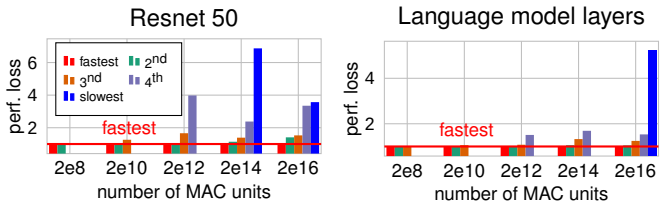


Fig. 13: Total runtime loss vs. best configuration for *scale-up* ie. aspect ratio (R:C). Colors differentiate configurations ordered by runtime.

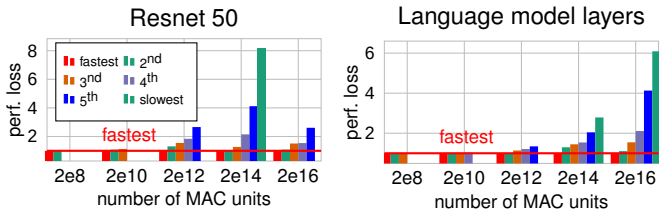


Fig. 14: Total runtime loss vs. best configuration for *scale-out*. ie part order (P_R, P_C) and aspect ratio (R:C). Colors differentiate configurations ordered by runtime.

locally optimal candidates for scale-out is depicted. In both these cases we observe that the pareto optimal configuration is up to 8x faster than the locally optimal configurations. However, the second and third best configurations are within 20% for smaller number of MAC in both scaled-up and scaled-out configurations. However as the MACs increase the spread of runtimes and we see about 50% increase un rutime for second and third best configurations, while slower configuration taking several factors more cycles to complete than the best configuration.

V. Related Work

Kwon et al. [15], propose a data centric model to determine the cost of a given dataflow over user specified accelerator configurations defined by a set of directives. This cost model is then used to search for an optimal dataflow. Timeloop [19] also uses a similar approach to determine the best mapping strategy, by analytical estimation of runtime and energy. Caffeine [31] describes analytical modelling of roofline performance to determine the hardware configuration for efficient FPGA-based CNN acceleration. Finally, ASV [6] constructs analytical energy/latency models of a given DNN and architectural configuration, while using constrained optimization to identify the best scheduling policy. However, all these works only consider a design space of monolithic accelerator configurations (scale-up).

DyHard-DNN [21] proposes the idea of morphable systolic arrays, with circuit techniques to save power. However unlike this work, the tradeoff space of partitioned designs is not investigated. ScaleDEEP [27] proposes a partitioned architecture, but includes heterogeneous compute units tailored for various types of workloads.

Tetris [8] describes a custom accelerator infrastructure, comprised of multiple partitioned units, implemented in the logic layer of a 3D memory. Tangram [9] extends the tiled accelerator architecture and adds extra functionality to improve compute and memory utilization by proposing custom dataflow for inter-layer pipelining and intra-layer reuse over a custom NoC. Moreover, the trade-off space for monolithic vs partitioned design is not exposed and explored in these papers.

Kung et al. [13] proposed a partitioned systolic array based solution. In this work the authors employ specialized 3D chip fabrication, and used through-silicon vias (TSVs) to mitigate the high memory bandwidth requirement. Simba [25] implements a scale-out accelerator using multi-chip modules (MCMs). The authors analyze the cost of scale-out infrastructure and propose a custom architecture comprised of MCMs accounting for the costs.

VI. Conclusions

In this paper, we analyze the various alternative approaches to Scale-up and Scale-out DNN accelerator designs. To conduct this study, we construct and describe a cycle accurate DNN accelerator simulator called SCALE-SIM. The simulation results provide memory accesses and bandwidth requirements for various layers of CNN and natural language processing model workloads for varying monolithic and partitioned systolic array based configurations. We also present an analytical model for performance estimation to chart and prune the search space of optimum configurations more rapidly. Our studies depict the inherent trade-off space for performance, DRAM bandwidth, and energy and identifies the sweet spots within the spaces for different workloads and performance points. We have open sourced our tool and hope that it benefits the community to conduct insightful studies as the one presented in this paper.

VII. Acknowledgements

We would like to express our gratitude to Abhinav Himanshu, Felix Kao, Sachit Kuhar, Vineet Nadella, and Natesh Raina for their help. Special thanks to Amrita Mathuriya for her insightful advice on scaling discussions. We would also like to express our gratitude the anonymous reviewers for their insightful comments and suggestions on improving the paper. Finally, we thank our open source contributors and users, for their pull requests, bug reports and constructive suggestions to improve SCALE-SIM. We look forward to keep improving SCALE-SIM in a healthy community driven fashion.

This work was supported by NSF CRII 1755876, a Google Faculty Award, and by a fellowship within the IFI programme of the German Academic Exchange Service (DAAD).

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] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International Conference on Machine Learning*, 2016, pp. 173–182.
[4] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *International Solid-State Circuits Conference*, ser. ISSCC, 2016.

[5] I. Fedorov *et al.*, "SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers," in *Advances in Neural Information Processing Systems 32 (NeurIPS)*, 2019, pp. 4978–4990.

[6] Y. Feng *et al.*, "ASV: Accelerated Stereo Vision System," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[7] J. Fowers *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 1–14.

[8] M. Gao *et al.*, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 751–764.

[9] M. Gao *et al.*, "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.

[10] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[11] X. He *et al.*, "Neural collaborative filtering," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 173–182.

[12] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[13] H. Kung *et al.*, "Maestro: A memory-on-logic architecture for coordinated parallel use of many systolic arrays," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 42–50.

[14] H. Kung *et al.*, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 821–834.

[15] H. Kwon *et al.*, "Understanding reuse, performance, and hardware cost of dnn dataflows: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 754–768.

[16] S. K. Lee *et al.*, "A 16-nm Always-On DNN Processor With Adaptive Clocking and Multi-Cycle Banked SRAMs," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1982–1992, July 2019.

[17] H. Li *et al.*, "On-Chip Memory Technology Design Space Explorations for Mobile Deep Neural Network Accelerators," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–6.

[18] S. Likun Xi *et al.*, "SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads," *arXiv e-prints*, p. arXiv:1912.04481, Dec 2019.

[19] A. Parashar *et al.*, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 304–315.

[20] M. Pellauer *et al.*, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 137–151.

[21] M. Putic *et al.*, "Dyhard-dnn: Even more dnn acceleration with dynamic hardware reconfiguration," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[22] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[23] K. S *et al.*, "Applications of Deep Neural Networks for Ultra Low Power IoT," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 589–592.

[24] A. Samajdar *et al.*, "Scale-sim: Systolic cnn accelerator simulator," 2018.

[25] Y. S. Shao *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*. ACM, 2019, pp. 14–27.

[26] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[27] S. Venkataramani *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 13–26, 2017.

[28] P. N. Whatmough *et al.*, "A 16nm 25mm2 SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators," in *2019 Symposium on VLSI Circuits*, June 2019, pp. C34–C35.

[29] P. N. Whatmough *et al.*, "FixyNN: Efficient Hardware for Mobile Computer Vision via Transfer Learning," in *Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA*, 2019.

[30] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[31] C. Zhang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[32] Y. Zhu *et al.*, "Euphrates: Algorithm-SoC Co-Design for Low-Power Mobile Continuous Vision," in *Proc. of ISCA*, 2018.