Partitioning FPGA-Optimized Systolic Arrays for Fun and Profit

Long Chung Chan, Gurshaant Malik, Nachiket Kapre
University of Waterloo
Ontario, Canada
lc6chan,gsmalik,nachiket@uwaterloo.ca

***Abstract*—**
We can improve the inference throughput of deep convolutional networks mapped to FPGA-optimized systolic arrays, at the expense of latency, with array partitioning and layer pipelining. Modern convolutional networks have a growing number of layers, such as the 58 separable layer `GoogleNetv1`, with varying compute, storage, and data movement requirements. At the same time, modern high-end FPGAs, such as the Xilinx UltraScale+ VU37P, can accommodate high-performance, 650 MHz, layouts of large 1920×9 systolic arrays. These can stay underutilized if the network layer requirements do not match the array size. We formulate an optimization problem, for improving array utilization, and boosting inference throughput, that determines how to partition the systolic array on the FPGA chip, and how to slice the network layers across the array partitions in a pipelined fashion. We adopt a two phase approach where (1) we identify layer assignment for each partition using an Evolutionary Strategy, and (2) we adopt a greedy-but-optimal approach for resource allocation to select the systolic array dimensions of each partition. When compared to state-of-the-art systolic architectures, we show throughput improvements in the range 1.3-1.5× and latency improvements in the range 0.5-1.8× against Multi-CLP and Xilinx SuperTile.

I. Introduction

Systolic arrays [1] organize hardware resources in a repeating grid of simple compute elements wired together using nearest-neighbour interconnect. The key idea is to inject data into the array in rhythmic fashion (to a systolic beat) and exploit data reuse through the nearest-neighbour connectivity. They can be configured to solve a variety of problems including matrix operations that are a common kernel in machine learning workloads. The hardware realization of these arrays is layout friendly and modern chips such as the Google TPU [2] have adopted this design style.

FPGA architectures are well-suited for efficient realization of 2D systolic arrays due to their regular arrangement of resources. Hard resources such as the Xilinx DSP48 math blocks, BRAM18 and URAM288 on-chip memories are laid out in a columnar fashion throughout the chip. Furthermore, Xilinx UltraScale+ devices naturally support systolic data movement using hard interconnect cascades along these columns. With careful floorplanning, it is easily possible to get 650 MHz+ operation [3], [4] on the Xilinx UltraScale+ VU9P–37P device(s).

A key limitation of mapping convolutional neural networks to systolic arrays is the threat of low array utilization.

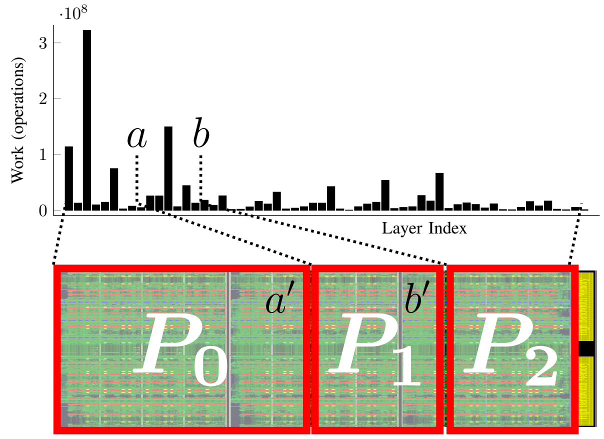


Figure 1: Illustrating the partitioning problem when mapping a deep neural network to a large systolic array. Here, we see the work across different layers of the `GoogleNetv1` mapped to a 1920×9 array on a Xilinx VU37P FPGA and split into three partitions. We need to choose split points a and b to compute layer assignment as well as a' and b' to distribute systolic array resources.

In a deep neural network, each layer has its own unique computational requirements and may be unable to use the full capacity the systolic array. As seen in Figure 1, the number of operations in each layer across the 58-layer `GoogleNetv1` topology varies quite dramatically. This mismatch can be remedied by tailoring the array size [5], [6] uniquely to each layer within the constraints of total chip capacity. Fortunately, the FPGA fabric naturally supports configuration opportunities that lets us **partition** or fracture the array as desired for each machine learning workload.

To use partitioned systolic arrays effectively, we need to split layers of the deep neural network across different sub-arrays on the device. The number of layers assigned to a partition and the size of the partition must both be chosen for maximizing utilization of hardware. This is a non-trivial problem due to the large number of layers in modern neural networks, and the large systolic array sizes that are possible on modern FPGAs. The solutions proposed in Multi-CLP design [5], [6] allow layers to be partitioned in an arbitrary manner complicating inter-layer data movement as well as creating a larger design space than necessary. The Xilinx SuperTile [3], [7] decomposes layers in contiguous subsets that capture inter-layer traffic within a partition and reduces the set of choices that need to made to a tractable level. Our approach follows the Xilinx SuperTile design but simplifies it further to only require 1D partitioning of the systolic array. We can visualize this task of splitting the layers and the physical systolic array resources in Figure [1]. For instance, we consider the case of contiguous partitioning on `GoogleNetv1` with 58 layers mapped to a Xilinx VU37P with a 1920×9 array. If we partition the network into K contiguous subsets, we have $57 \times 56 \times 55 \times \ldots (57-K-1)$ possible partitions. Similarly, we can split a 1920×9 array along the first dimension (1D partition) into K contiguous sub-arrays in $(1920-K-1) \times (1920-K-2) \times (1920-K-3) \times \ldots (1920-2K-1)$ possible ways. Thus, the total space of choices is the product of the two terms which can quickly become infeasible to naïve brute-force search.

In this paper, we develop a fast optimization algorithm to contiguously partition the neural network across a systolic array to improve array utilization. We generalize the problem to arbitrary number of partitions, target a broader set of neural networks, and do so using evolutionary algorithms to guide the search. We use three strategies, two of which are inspired by evolutionary algorithms, to attack the partitioning problem: CMA-ES (Covariance Matrix Adaptation Evolution Strategy), GA (Genetic Algorithm), and Hyperopt [8] (Hyper-parameter optimization). These algorithms use an iterative approach for discovering solutions and are able to generate high-quality partitions in a few seconds.

The key contributions of this paper include:

- Formulation of an optimization problem for partitioning FPGA-optimized systolic arrays to improve their utilization when mapping deep convolutional networks.
- Development of a two phase approach to compute (1) layer assignment using a search process, and (2) resource allocation using a greedy-but-optimal approach. This formulation makes the problem tractable.
- Use of SCALEsim systolic array modeling framework to generate a cycle-accurate performance model for use with the optimization flow.
- Comparison of two evolutionary strategies CMA-ES and GA with an off-the-shelf parameter tuning framework Hyperopt for solving the optimization problem.
- Quantification of the throughput-latency trade-offs, optimization runtime, for benchmarks derived from the MLPerf [9] dataset and other ConvNets.

II. Background

We first describe how our systolic arrays are mapped to an FPGA and then discuss evolutionary algorithms.

A. Systolic Arrays for CNNs on FPGAs

Systolic data movement is crucial for exploiting abundant data reuse opportunities in deep neural networks. A modern Xilinx UltraScale+ VU37P FPGA supports 960 URAM blocks, 9024 DSP48 slices, and 4032 RAMB18 blocks. We can build a systolic overlay of size 1920×9 where each row is a chain of 9 DSP48 blocks. We use the design from [4] in this work, where you can find a detailed discussion of the FPGA-optimized systolic array implementation. The length-9 chain is chosen to fit a 3×3 convolution while using the DSP48 cascades in the computation core. In this architecture, a pair of URAMs supplies data to 2×9 array of systolic multiply-add blocks mapped to SIMD=2 DSP48 units. By carefully staging data through the URAMs, BRAMs, and internal DSP A+B registers, we can orchestrate systolic behavior from the components. For matrix-vector multiplication, we are performance limited by the memory bandwidth of the URAM blocks. This halves the effective array size available for those layers. As seen in Figure [2], the DSP-to-DSP links form one (horizontal) dimension of the systolic array for both convolution and matrix-vector processing. The 72b URAM cascades provide an equivalent systolic lane support in the vertical dimension. For matrix-vector processing, we only need to redistribute the result vector in a systolic fashion for the next layer. For the large VU37P, the URAM capacity is large enough to hold all the weights and worst-case activations for networks like `GoogleNetv1`. For those designs where that is not possible, the 32× 256b AXI connections to a multi-ported on-chip HBM memory bank permits rapid loading of the URAM memory structures.

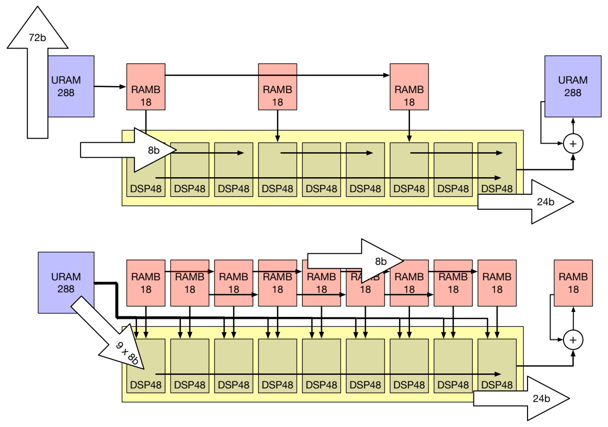


Figure 2: Systolic building blocks for Convolution and Matrix-Vector multiplication that exploit nearest neighbour data movement in weight-stationary and input-stationary manner. A cascade of 9 DSP48 blocks is the minimum repeating unit that is replicated across the chip.

B. Neuro Evolution

The *Neuro-Evolution* (NE) *ethos* proposes the use of evolution-based algorithms to solve difficult optimization problems. In an NE algorithm, candidate solutions are refined through a series of evolution step (generations) that teach the algorithm how to nurture desired characteristics. In one step, a set of mutations are performed on to produce an ensemble of potential solution models. Each of these potential model is then evaluated for fitness specific to the optimization task, followed by a fitness-ranked selection and evolution of best-performing models into the parent set for the next generation of evolution. Reinforcement Learning (RL) [10] and Neural Network topology search for classification problems [11] have seen successful implementations of NE in recent works. NE is particularly effective for applications where computation of gradients are intractable. The efficacy of NE techniques is mainly due to the evolution mechanism that reliably discovers and nurtures desired model characteristics and suppressing detrimental ones. We discuss two broad categories of NE-based algorithms:

Evolution Strategies (ES): Evolution Strategies [12], [13] discover problem structure by representing the candidate solution as a distribution of random variables. At every generation, candidate solutions are generated using this distribution and evaluated for their fitness on the task being learned. The top performing candidates are selected via *deterministic survivor selection* and this is used to evolve the distribution representative of the solution space.

Genetic Algorithms (GA): GAs [14]–[16] aims to accurately mimic biological evolution by mapping the space of problem variables to a genome. Through the course of the evolution cycle, GAs use mutation and crossover of genomes to produce a set of competing and diverse candidate models. Each phase of *mutation-crossover* is seeded from the best performing candidates from previous generations.

III. Partitioning Algorithm

First, we motivate the need for partitioning systolic arrays for deep networks with an example that demonstrates the scale of underutilization possible in an array. Next, we formalize the objective of our partitioning algorithm and illustrate the working of one algorithm on a simple example.

A. Motivation

Large, monolithic systolic arrays of dimensions 1920×9 are now easily realizable on modern Xilinx UltraScale+ FPGAs. When mapping layers of a deep network to such an array, performance if often limited by the amount of parallelism in that layer and its memory bandwidth requirements. To concretely observe these trends, in Figure 3, we show cycle count and array utilization for the `Conv1` layer of `GoogleNetv1` neural network when the systolic array size is varied from 1×9 to 1920×9. As expected, when the array size increases, we see an improvement (reduction) in cycle count required to process the array. However, if we provide resources beyond a certain limit, 60× 9 in the example, there is negligible improvement in performance and most of the array remains idle. If layers of a deep network are serially processed and the entire systolic array is made available to each layer, we will observe massive underutilization and loss in throughput. Instead, we can repurpose the idle portion of the array to process other layers of the network in a pipelined fashion [3], [5]–[7].

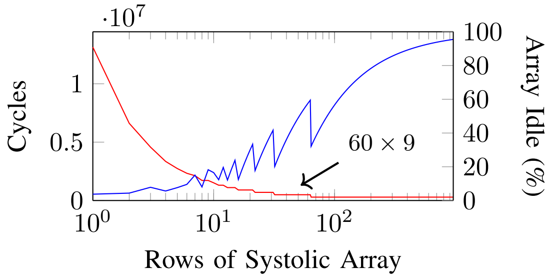


Figure 3: Cycle count and array utilization scaling trends for `GoogleNetv1 Conv1` layer. As we scale beyond 60×9 array size, the array idle time grows beyond 50% fast approaching 90% at full system size of 960×9. The noise in array utilization is due to the quantization effect of managing reuse.

Layer pipelining, like classic datapath pipelining, allows a design to start computation of a next input image on an early layer of the network, while a previous image is still being processed in downstream layers of the network. While this transformation may compromise latency, it will let hardware resource stay busy with useful work, thereby improving inference throughput. In this paper, we formulate the partitioning problem in more general terms that (1) works for any network and any systolic array size, (2) provides finer-grained partitioning support down to individual row granularity, and (3) integrates with a fast neuroevolution algorithm to discover high quality partitions.

B. Optimization Formulation

The objective of our mapping algorithm is to assign contiguous non-overlapping subsets of neural network layers to physically-disjoint 1D partition of the systolic array. A brute force search of possible solutions is intractable for deep networks and large array sizes. For an N-layer network mapped to a 1920×9 array split into K partitions, we can formalize the objective function we wish to minimize as shown below:

$$\min_{l,p}\left(\max_{x}\left(\sum_{y\in l[x]} cycles[y][p[x]]\right)\right) \quad (1)$$

$$\text{subject to} \sum_{x=0}^{K-1} p[x] = 1920 \quad (2)$$

$$\sum_{x=0}^{K-1} l[x] = N \quad (3)$$

$$\forall x, p[x] \geq 1 \quad (4)$$

$$\forall x, l[x] \geq 1 \quad (5)$$

In Equation 1,

- x is the partition index,
- $p[x]$ is the size of systolic array for that partition,
- $l[x]$ is the set of layers mapped to the partition, and
- $cycles[][]$ is the timing model for the systolic array implementing a particular neural network topology.

To solve this equation, we first construct an empirical timing model captured by the 2D array $cycles[][]$ for each layer of a neural network. This array is indexed first by the layer index y of the layer mapped to all possible systolic array sizes from 1×9–1920×9. This array is built from a cycle-accurate simulation of the RTL design and the DRAM interface using the SCALEsim [17] modeling framework. For a partition x, we must add up the cycles needed per layer mapped to that partition (array $l[x]$). This is because within a partition the layers are executed sequentially. Across all partitions, the computation is pipelined, which means the overall system throughput is defined by its slowest partition. Thus, we can compute a $\max$ of the cycles required by each partition and this figure is the object of optimization minimization. This measurement is analogous to critical path analysis in determining clock frequency of RTL designs. Our optimization algorithm will aim to discover a layer assignment $l[x]$ and associated resource allocation $p[x]$ to minimize the worst-case cycle count across all partitions. This is captured by the objective function in Equation 1. A legal solution must ensure non-empty layer assignments (Equation 4) and non-empty partitions (Equation 5). Choosing the values of $l[x]$ determines the values a and b in Figure 1 while selecting $p[x]$ implies determination of a' and b' in the same example. Unlike Multi-CLP [5], we do not constrain weights and activations to fit within on-chip capacity as we rely on our optimizer to discover the best strategy.

C. Search Algorithm Design

The key idea we use to constrain the search problem is to split the process into two steps (1) layer assignment, and (2) resource allocation. This allows the search complexity of layer assignment to be decoupled from resource allocation. Furthermore, this allows the resource allocation step to be computable in polynomial time. The layer assignment process is handled by an intelligent search algorithm (CMA-ES, GA, of Hyperopt). Once we know which set of layers are assigned to which partition $l[x]$, we can determine resource allocation $p[x]$ in a greedy, optimal manner. This is possible because (1) we already know that the cycle count scaling trends for each layer in the $cycles$ array are monotonically decreasing as a function of systolic array size, and (2) we are only interesting in minimizing the maximum cycle count across all partitions. The complete process is illustrated in Algorithm 1.

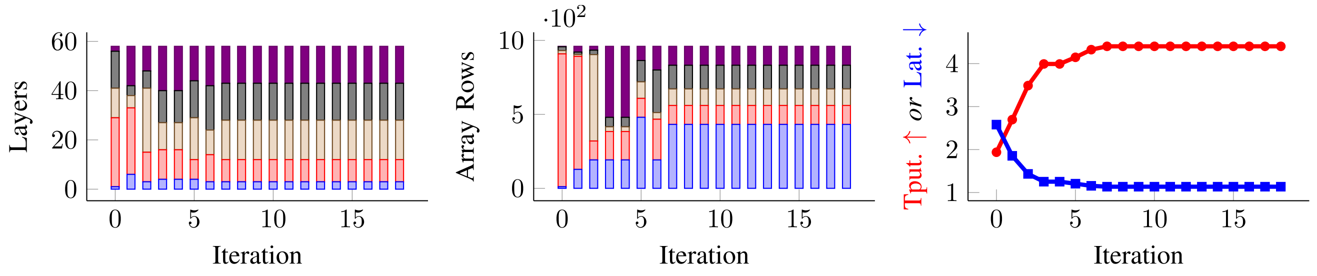
We illustrate the operation of this search algorithm using Figure 4 for `GoogleNetv1` mapped to a 1920×9 array (max) with 5 partitions. As the evolutionary algorithm proceeds, the solutions start to change before settling down into stable values after ≈6-7 iterations. In each iteration, the mutation step generates multiple candidate solutions, evaluates their cost functions, and learns which combinations work well and which fail. This results in a steady improvement in resulting throughput at the expense of increased latency. At steady state, the first partition (at the bottom of the stacked bar chart) captures the first few layers of `GoogleNetv1` while getting almost 50% of the resources. A cursory glance at the operation count distribution from Figure 1 confirms this is an intuitively correct solution. Other workloads like `AlphaGoZero` have stubborn layers with limited parallelism, and require introspection into the parallelizability, and memory capacity + bandwidth constraints of the layer to correctly determine the layer partition and resource assignments.

Algorithm 1: Simplified View of the layer assignment and resource allocation algorithm to compute $p[x]$ in the inner loop while exploring layer assignment $l[x]$ using intelligent search for the outer loop.

```
while !terminate do
    $l[x]$ = New_Candidate(); iteration++;
    for $x < K$ do
        // Start with $1\times9$ arrays in each partition
        $p[x]$ = 1
        // Compute cycles spent in partition $x$
        $cyc[x]$ = $\sum_{y\in l[x]} cycles[y][p[x]]$
    // Loop until resources $R$ still available
    R = 1920-K;
    while $R > 0$ do
        // Find the bottleneck partition
        $x'$ = $\max_x(cyc[x])$
        // Increase allocation to bottleneck partition
        $p[x']$++; R=R-1;
        // Recalculate cycles for partition $x'$
        $cyc[x']$ = $\sum_{y\in l[x']} cycles[y][p[x']]$
    $cost$ = $\min_{l,p}\max_x \sum_{y\in l[x]} cycles[y][p[x]]$
    // Provide training data to search algorithm
    Train($l[x]$,$cost$)
```

IV. Experimental Setup

We show a high-level diagram of our toolflow in Figure 5. We do a one-time construction a timing (performance) model of the systolic array of specific dimensions for a particular neural network topology by sweeping each layer of the network across various systolic array sizes. We then run an optimization loop that first determines the layer assignments using an Evolutionary Strategy while deciding resource allocation using a greedy-but-optimal approach.



(a) Layer Assignment $l[x]$ (b) Resource Allocation $p[x]$ (c) Throughput-Latency tradeoffs

Figure 4: Evolution of layer assignment $l[x]$ and resource allocation $p[x]$ for `GoogleNetv1` with 5 partitions mapped to a Xilinx UltraScale+ VU37P FPGA with a 1920×9 array (max). Iteration refers to the outer `while` loop in Algorithm 1.

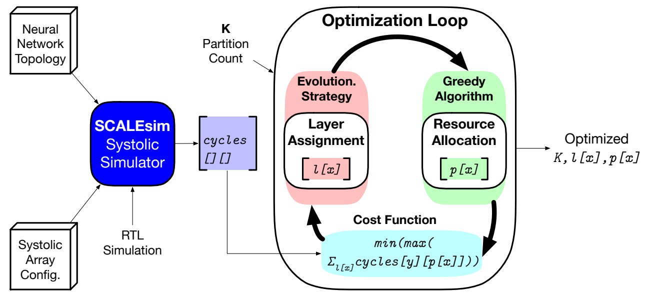


Figure 5: High-level diagram of the partitioning toolflow. SCALEsim builds the performance model $cycle[][]$. The evolutionary + greedy algorithm finds layer assignment $l[x]$ and resource allocation $p[x]$ for a partition size K.

A. FPGA Design

We construct the FPGA-optimized systolic array hardware using direct instantiation of Xilinx hard blocks such as DSP48, RAMB18, and URAM288. We design state machine controllers to manage data movement between the various resources and provide partitioning support by gating data movement in the URAM and BRAM chains. We floorplan the design using Xilinx XDC constraints and implement it on a Xilinx UltraScale+ VU37P FPGA. We use the design from [4] that is able to fit a systolic array of size 1920×9 in this chip and operate it at a high 650 MHz clock frequency limited solely by the URAM maximum operating frequency. The specific cycle counts needed by our hardware array are extracted from RTL simulation of the building blocks and provided to the performance modeling tool for allow large-scale experiments on various topologies and system sizes that would otherwise be too slow for RTL simulations.

B. Performance Modeling

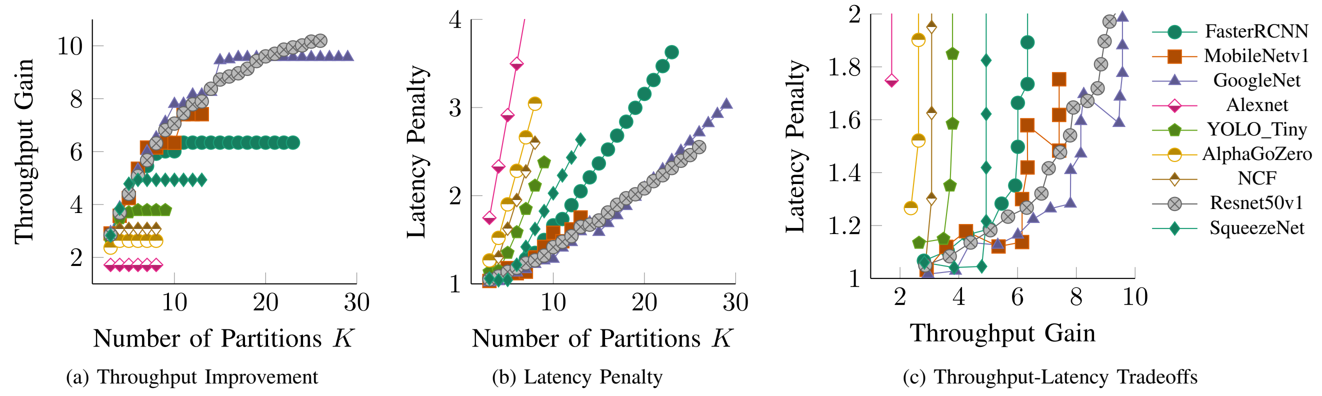
To realize our optimization algorithm, we need to generate the $cycle[][]$ timing model for each convolutional network topology at various systolic array sizes. We compute the cycle counts needed by each layer of the network topology individually by exploring all design combinations between 1×9 and 1920×9 sizes in steps of one. We use the SCALEsim [17] systolic array modeling framework that supports the flexibility of evaluating different styles of dataflow as appropriate for convolution and matrix-vector processing stages. We configure SCALEsim to account for memory capacity limits of the URAM as a second stage of memory in the architecture. We supply data to the URAMs from the multi-ported high-bandwidth HBM memory and capture the correctly, as smaller array sizes only have access to a proportionally reduced number of URAM resources which affects capacity and increases pressure on the DRAM interfaces. Thus, smaller systolic arrays require higher cycle counts due to combination of two factors including fewer resources for exploiting parallelism and reduced memory bandwidth to supply data. We use benchmarks from MLPerf [9][1] and other ConvNets from Multi-CLP [5].

C. Evolutionary Algorithms

We compare the effectiveness of two kinds of evolutionary algorithms in this paper that optimize the throughput-oriented goodness metric in Equation 1 via Algorithm 1.

CMA-ES: The first style is CMA-ES (Covariance Matrix Adaptation Evolution Strategy) where the unknown real-valued variables are modeled as Gaussian distributions with a mean and variance. An evolution step involves generating a population of solution candidates and evaluating their cost functions. At the end of an evolution step, the top 25% of the best solutions are retained and used to update the mean and variance of each unknown. In our implementation, each variable represents a percentage of layers included in that partition. For example, an array of $[0.2, 0.3, 0.5]$ can be decoded as have the first 20% of layers in the first partition, the next 30% layers in the second and the remaining 50% layers in the final partition. We initialize the system in either (a) all-zero assignment, or (b) valid legal assignment, for $l[x]$. We reject illegal assignments with high penalty. An illegal assignment happens when either $l[x]$=0 for any x, $l[x_a] < l[x_b]$ for $x_a > x_b$. These two cases capture the condition where there is an empty partition or the layer assignment starts at a layer index larger than where it ends (an impossibility).

[1]Result not verified by MLPerf.



(a) Throughput Improvement (b) Latency Penalty (c) Throughput-Latency Tradeoffs

Figure 6: Understanding the impact of partitioning on Throughput and Latency across MLPerf and ConvNet workloads.

GA: We also evaluate the effectiveness of Genetic Algorithm approach that naturally supports integer solutions. Unlike CMA-ES, GAs can directly manipulate integer unknowns. In our implementation, we use a permutation GA approach that select K-1 partition split points from a random permutation of vector $[1, 2, \ldots, N-1]$ where N=number of layers and K=number of partitions. For example, with N=6 and K=3, we select 2 split-points from the first two locations of the vector and ignore the rest. If the gene has values $[\underline{3,1},4,5,2]$, we will split after first and third layers to generate three partitions. An advantage of this problem formulation is that it is guaranteed any off-spring generated using mutation will be a valid solution.

Hyperopt: Finally, we investigate a flavour of Sequential model-based Bayesian optimization using Hyperopt [18]. Hyperopt is able to handle integer-valued search variables with ease. As a result, in similar style to GA, we are able to define a *configuration space* assignment to directly optimise for $l[x]$ without numerical reshaping. We configure Hyperopt to use the `Tree-of-Parzen-Estimators` (TPE) algorithm [19] to optimise the search space.

V. Evaluation

We now investigate the use of our partitioning algorithm on the resulting performance improvements on the systolic array. We measure inference throughput (img/s), end-to-end latency as a function of various experiment parameters such as number of partitions K, choice of optimization algorithm, and optimization time. We also examine the quality-time tradeoffs in choice of evolutionary algorithms we use.

A. Throughput and Latency Tradeoffs

In Figure 6, we explore the effect of varying partition sizes on the resulting inference throughput and latency of the neural network. We compute throughput gain and latency penalty compared to a non-partitioned baseline where the entire array is allocated to each layer of the neural network. As we increase the number of partitions of the network, we note an improvement in throughput due to an associated increase in systolic array utilization. Beyond a certain partition count, we no longer observe any increase in throughput due to saturation of compute resources and memory bandwidth. The exact threshold where this happens varies with the workload. For instance, for large networks like `GoogleNetv1`, we observe throughput wins of $\approx$10$\times$ at 15 partitions at the expense of 1.3$\times$ increase in inference latency. Other networks like `FasterRCNN` saturates earlier at around 6–7 partitions and delivers proportional throughput improvements of $\approx$6$\times$. In the extreme end, shallow networks with 5–10 layers like `AlphaGoZero` and `AlexNet` only show limited throughput improvements of 2–3$\times$ and only scale to limited partition counts.

We can also visualize the relative effects of changing partition size on both throughput and latency together as shown in Figure 6c. We clearly observe the almost linear relationship between throughput improvements and increase in latency of inference. Bulk of the explored design configurations only slow down inference by 2$\times$ but are able to deliver as much as 10$\times$ throughput improvements. Higher latency penalties as seen previously in Figure 6, happen when the design configurations are *overpartitioning* the systolic array that are dominated by strictly superior solutions.

Finally, in Figure 7, we report a Figure of Merit (FoM) score which is computed as the ratio of Throughput gain to Latency loss for each workload. As we increase partition count, we have seen that throughput gains increase at the expense of latency losses. The ratio captures a sweet spot that can be achieved where we achieve substantial improvements in throughput without sacrificing too much latency. The smallest partition size where this can be done is then reported as the ideal partition size for that workload. For benchmarks like `GoogleNetv1`, and `Resnet50v1`, we can scale to 9–10 partitions at peak FoM value. For medium-sized benchmarks like `FasterRCNN`, `SqueezeNet`, and `MobileNetv1`, we can scale to 6–8 partitions, while shallow networks like `AlphaGoZero`, `YOLO_Tiny` and `AlexNet`, scale to 3-4 partitions.

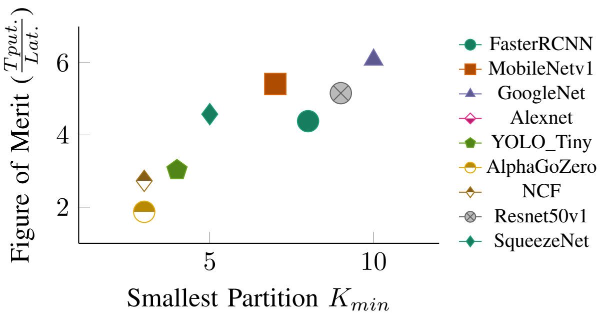


Figure 7: Best Figure-of-Merit (FOM) score for each workload and associated smallest partition count K_{min}.

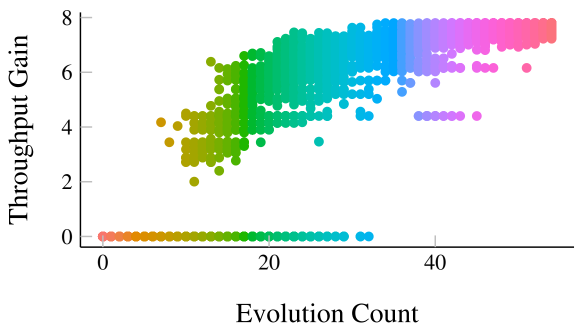


Figure 8: Improvement in Throughput for `GoogleNetv1` for 10 partitions across iterations of Evolutionary Strategy.

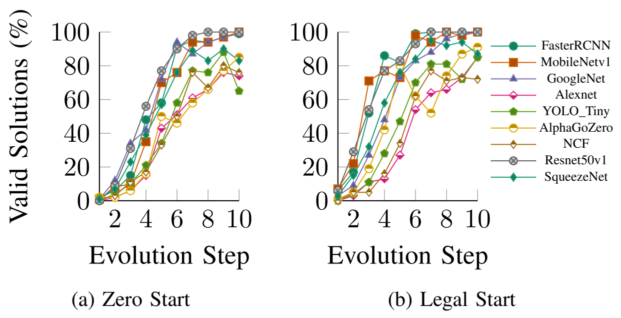


Figure 9: Tracking the valid solutions percentage as a function of evolution step across various workloads for 5 partitions.

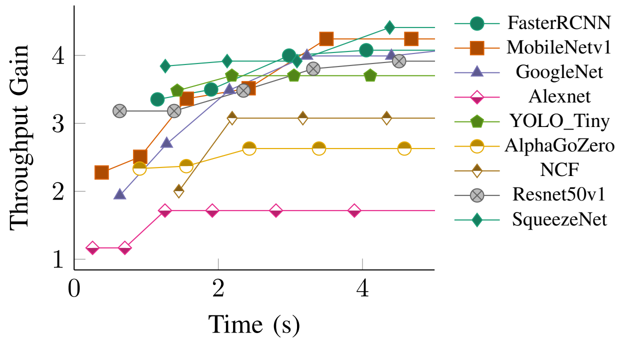


Figure 10: Improvement in solution quality as a function of time for various networks with 5 partitions.

B. Understanding Evolution

We now look how the Evolutionary Strategy (CMA-ES) helps navigate us towards the solution in Figure 8 for `GoogleNetv1` with 10 partitions. Here, we see that the partition solutions *evolve* towards better throughput and ultimately saturate at a speedup of 8× after ≈30 iterations. Each iteration operates on a population size of 100 mutations to determine the direction of evolution. The extent of solution quality also improves with the evolutionary algorithm iterations. The colors indicate the evolutionary process as the system converges towards the best partitioned solution. We initialize our system with an illegal all-zero solution vector for network layers and partition sizes and reject those combinations with high penalty. After the first ≈25 iterations, the optimizer has *learned* enough about the solution space to no longer generate illegal candidates as we see with the gain=0 cluster at the bottom of the plot.

We can also track the learning process in CMA-ES by inspecting Figure 9 (valid solutions) and Figure 10 (runtime). In Figure 9, we see the effect of zero start (illegal solution) slows down convergence by a small amount 1–2 evolutionary steps. In each step, we explore 100 mutations, and some of those mutations are illegal. With a legal starting condition, the different networks are able to observe exploration of 80% valid solutions between 3–4 iterations while a zero start delays this to 5–6 iterations. Both cases are ultimately able to discover the best partitioning strategy and deliver identical throughput and latency wins. When considering runtime required in Figure 10, we note that the our evolutionary strategy completes the search in a few seconds of exploration! This illustrates the speed and robustness of the evolutionary strategy to absence of domain knowledge in seeding the search process.

Finally, in Figure 11, we show the differences in runtime for a select few benchmarks across CMA-ES, GA, and Hyperopt search algorithms for 8-partition problems. Hyperopt runs quickly but typically settles for a lower quality result as it is trapped in a local minima. GA generates various permutations, slowing it down, and also resulting in a lower quality of result. CMA-ES shines across the board with a higher quality solution in all cases. For certain workloads like `NCF` (and `AlexNet`, `AlphaGoZero` now shown) the design space is small enough that the search completes in the first iteration itself.

C. Comparison against state-of-the-art

Finally, in Table I, we position our work against previously best-published performance data for Multi-CLP [5] and Xilinx SuperTile [3]. The `AlexNet` and `SqueezeNet` topologies are realized in 32b and 16b floating point precision respectively on the Multi-CLP architecture which can constrain the largest systolic array you can fit on the device.

Table I: Comparing the throughput and latency of state-of-the-art FPGA systolic arrays like Multi-CLP and SuperTile.

Topology	FPGA	K	Array Size (×9 for This paper design)	Throughput (img/s)	Latency (ms)
`AlexNet` [5]	485T	4	[2x64 1x96 3x24 8x19]	61	62.3
`AlexNet` [This Paper]	485T	3	[10 30 10]	90 (1.47× ↑)	42.7 (1.45× ↓)
`AlexNet` [5]	690T	6	[1x64 1x96 2x64 1x48 1x48 3x64]	80	70.1
`AlexNet` [This Paper]	690T	4	[11 42 9 2]	102 (1.27× ↑)	37.8 (1.85× ↓)
`SqueezeNet` [5]	485T	6	[6x16 3x64 4x64 8x64 8x128 16x10]	913	6.52
`SqueezeNet` [This Paper]	485T	6	[54 32 43 24 64 32]	1166 (1.27× ↑)	5.12 (1.27× ↓)
`SqueezeNet` [5]	690T	6	[8x16 3x64 11x32 8x64 5x256 16x26]	1167	5
`SqueezeNet` [This Paper]	690T	17	[96 3 8 4 16 43 3 11 20 7 16 6 4 30 3 6 34]	1579 (1.35× ↑)	10.62 (2.1× ↑)
`SqueezeNet` [This Paper]	690T	8	[96 13 13 43 32 24 43 46]	1429 (1.22× ↑)	5.57 (1.14× ↑)
`GoogleNetv1` [3]	VU9P	12	[21x8 32x16 96x16 8x1] × 3	3046	3.9
`GoogleNetv1` [This Paper]	VU37P	14	[32 64 22 15 32 27 16 9 19 24 20 16 18 6] × 6	5976 (1.9× ↑)	14.1 (3.6× ↑)
`GoogleNetv1` [This Paper]	VU37P	15	[64 7 192 64 43 96 48 56 38 43 75 80 64 64 26] × 2	4312 (1.4× ↑)	7.26 (2× ↑)

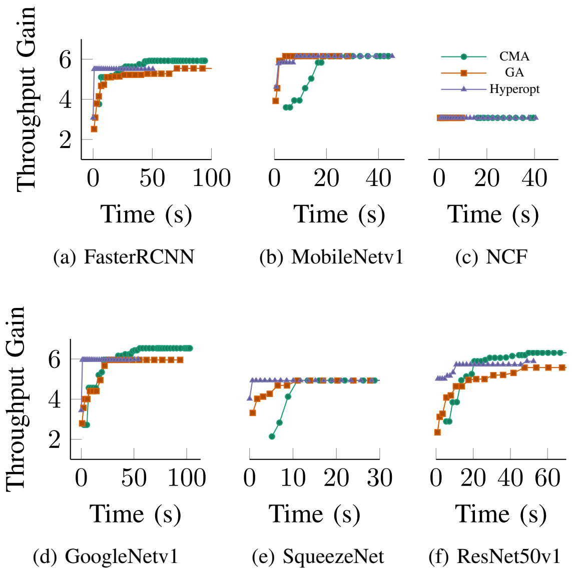


Figure 11: Exploring Throughput Gain trends for various parameter tuning algorithm for 8 partitions.

To ensure fair comparison, we reduce the size of our array to match their design specification. For the Xilinx SuperTile design, performance is reported on the Xilinx UltraScale+ VU3P which approximately accommodates an array of size 320×9. From the table, we note that we outperform Multi-CLP by 1.3–1.5× in terms of throughput and by 1.4–4.6× in terms of latency, while beating the throughput of SuperTile by 1.5× but have a 2× higher latency due to throughput-maximizing 15 partition solution (See Figure 7 to balanced designs). These solutions are possible due to the ability to explore a larger search space of possible solutions.

VI. Related Work

The Multi-CLP architecture presented in [5], [6] has explored the problem of partitioning FPGA resources across layers of a neural network. In that design, the problem has been made more complicated than strictly necessary by allowing (1) per-partition sizing of systolic array dimensions, and (2) arbitrary layer assignment without contiguity. The formulation faces from following challenges:

- Resource and runtime overheads of inter-partition communication. As layer sequence is not localized to a partition, we must explicitly move intermediate results which will cost us time (which may be partially overlapped) and FPGA resources.
- FPGA layout challenges for fitting multiple partitions with arbitrary sizes that do not compose in a 2D rectangular fashion. This will impact implementation frequency of the design.
- Discarded solutions due to on-chip memory capacity constraints for determining feasibility of layer assignment to partition. This may prematurely eliminate solutions that may have been adequate with some time overhead of fetching excess content from DRAM.

The Xilinx SuperTile [3] design offers a one-off multi-processor solution for `GoogleNetv1` mapped to four custom-sized systolic arrays. The layer assignment and partition sizing is done for this problem alone and no general solution is provided for arbitrary networks or chip capacities. Despite this limitation, we found SuperTile to be quite competitive with our eventual solution discussed in Table I.

VII. Conclusions

In this paper, we show how to boost inference throughput of deep networks mapped to FPGA-optimized systolic arrays. We are able to outperform state-of-the-art Multi-CLP architecture by 1.3–1.5× on throughput and 0.5–1.8× on latency, provide 1.4× throughput over Xilinx SuperTile at the cost of 2× higher latency while consuming identical systolic resources. We demonstrate the use of an evolutionary algorithm CMA-ES to tackle a two-phase formulation of the partitioning problem. We offloads 1D layer assignment in the first phase to CMA-ES while using a greedy-but-optimal resource assignment strategy in the second phase. We observe that CMA-ES delivers higher quality solutions and also successfully bootstraps from a zero start requiring no a priori knowledge of the design space.

Source Code: https://git.uwaterloo.ca/watcag-public/fpga-syspart

References

[1] Kung, “Why systolic architectures?” *Computer*, vol. 15, no. 1, pp. 37–46, Jan 1982.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 1–12.

[3] E. Wu, X. Zhang, D. Berman, I. Cho, and J. Thendean, “Compute-efficient neural-network acceleration,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, 2019, pp. 191–200. [Online]. Available: https://doi.org/10.1145/3289602.3293925

[4] A. Samajdar, T. Garg, T. Krishna, and N. Kapre, “Scaling the cascades: Interconnect-aware fpga implementation of machine learning problems,” in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 1–8.

[5] Y. Shen, M. Ferdman, and P. Milder, “Maximizing cnn accelerator efficiency through resource partitioning,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 535–547. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080221

[6] Y. Shen, M. Ferdman, and P. Milder, “Overcoming resource underutilization in spatial cnn accelerators,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–4.

[7] E. Wu, X. Zhang, D. Berman, and I. Cho, “A high-throughput reconfigurable processing array for neural networks,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.

[8] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox, “Hyperopt: a python library for model selection and hyperparameter optimization,” *Computational Science & Discovery*, vol. 8, no. 1, p. 014008, 2015. [Online]. Available: http://iopscience.iop.org/article/10.1088/1749-4699/8/1/014008

[9] “Mlperf: Mlperf benchmark suite,” https://github.com/mlperf, 2018.

[10] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, “Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning,” *arXiv preprint arXiv:1712.06567*, 2017.

[11] A. Gaier and D. Ha, “Weight agnostic neural networks,” 2019.

[12] N. Hansen, “The cma evolution strategy: A tutorial,” *arXiv preprint arXiv:1604.00772*, 2016.

[13] D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber, “Natural evolution strategies,” in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 2008, pp. 3381–3387.

[14] L. Davis, “Handbook of genetic algorithms,” 1991.

[15] D. Whitley, “A genetic algorithm tutorial,” *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[16] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[17] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, “Scale-sim: Systolic CNN accelerator,” *CoRR*, vol. abs/1811.02883, 2018. [Online]. Available: http://arxiv.org/abs/1811.02883

[18] J. Bergstra, D. Yamins, and D. D. Cox, “Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms,” in *Proceedings of the 12th Python in science conference*. Citeseer, 2013, pp. 13–20.

[19] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” in *Advances in neural information processing systems*, 2011, pp. 2546–2554.