Implementing FPGA-optimized Systolic Arrays using 2D Knapsack and Evolutionary Algorithms

by

Long Chung Chan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

Statement of Contributions

This work is an extension of my own work published in Chan et al. (2019) [12].

The initial idea of incorporating CMA-ES in optimization of the partitioning workflow was implemented with the help of Gurshaant Malik.

The following 2D knapsack optimization is implemented with the help of Justin Borromeo.

The systolic array framework is provided by Thomas Kidd and Prof. Nachiket Kapre.

Abstract

Underutilization of FPGA resources is a significant challenge in deploying FPGAs as neural network accelerators. We propose an FPGA-optimized systolic array architecture improving the CNN inference throughput by orders of magnitude compared to an un-partitioned systolic array through parallelism-aware partitioning of on-chip resources. We fracture the FPGA into multiple square systolic arrays and formulate the placement of these arrays as a 2D knapsack problem. We simulate the cycle counts needed for each neural network layer given different systolic array sizes using cycle-accurate systolic array simulator - SCALESim. We generate physical implementation and operating frequencies of systolic arrays placed in uniformly staggered locations on Xilinx VU37P and VU9P Ultrascale+ platforms. We use the cycle and frequency information in an optimizer coupling CMA-ES evolutionary algorithm and a simple 2D Knapsack solver to discover packable and routable partitioned designs to maximize throughput. Our experiments' most significant performance improvement comes from the implementation of layers with large kernel sizes. We demonstrate that inference throughput gain of 7-22.7× is possible with a 1.2-7.6× sacrifice of latency. Our optimization tool can achieve up to 8× higher throughput gain on eight MLPerf benchmark network topologies. Our tool also generates designs across various latency and throughput combinations, providing a wide degree of design selection.

Acknowledgements

I would like to thank everyone in the WatCAG lab. 2019 and 2020 have made everything different and posed a new challenge to everyone while forcing everyone to adapt to this weird era as soon as possible. Luckily, we have weekly group meetings to share and discuss our research progress. Justin Borromeo is incredible to work with, and his fantastic data handling skills have allowed us to summarize our results quickly.

Next, I would like to thank Prof. Nachiket Kapre. Taking the ECE327 course taught by Nachiket back in Fall 2018 had changed my career path in a way I could not have imagined. Without Nachiket's support, I would not have had the chance to present my paper in ICFPT 2019 or the experience of being a master's student. Without his drive and guidance, I could not have finished this work within the timeframe. I am thankful for all the unique research experiences under Prof. Nachiket Kapre.

Finally, I would like to thank my family and friends for their support. On numerous occasions, they encouraged me to move forward during 2021. Even during this stressful year, they were always there to help and give me endless encouragement. I would not have emotionally survived this pandemic without them.

Dedication

This is dedicated to the unconditional support and love from my family.

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

With the slowdown of Moore's law of technology [32, 2], transistor size reduction has plateaued while the demand for computing power becomes more significant than ever. In recent years, the rapid adoption of convolution neural networks has allowed machine learning to be adopted across various fields, including recommendation systems, classification tasks, and natural language processing [28, 16, 6]. Due to the unique power-efficient and programmable computing substrate, Field Programmable Gate Arrays (FPGAs) have emerged as a potential candidate for post-Moore's law computing in data centers where high throughput operation with low power consumption is a significant concern [9, 42, 15]. Over the past decade, FPGA chips have become a popular choice of accelerators or specialized processors [15] in enormous data centers.

FPGA vendors have implemented various features in both software and hardware sides targeting domain-specific applications where one such area is artificial intelligence (AI). Due to the growing interest in using FPGAs as machine learning accelerators, FPGA vendors have fine-tuned the underlying fabrics to increase the computation capability of their products. One of the significant changes is more specialized hard blocks to improve inference performance. Intel introduced Stratix 10 NX, a 14nm FPGA platform including AI tensor blocks allowing 143 int8 TOPS at its peak performance in the fabric, allowing comparable performance to 12nm GPUs [11]. The AI tensor blocks on this platform are specially tuned arrays of lower-precision multipliers targeting throughput focused AI inference applications [11] offering comparable performance to GPU. Xilinx Ultrascale+ platform features hard blocks such as DSP48, BRAM18, and the URAM288. On VU37P and VU9P, Xilinx includes three Super Logic Regions (SLR) connected via Super Long Line (SLL) routing allowing each SLR to communicate with its neighboring SLR. Xilinx also introduces Versal FPGAs [62] in 2020, featuring an AI engine with an array of VLIW SIMD high-performance processors aimed at 5G and DNN applications.

However, it is still up to the applications engineers to utilize all these resources on the FPGA device to develop an efficient implementation for processing the computation of a neural network. The most dominant workloads in a Convolution Neural Network (CNN) are convolution operations with different filter dimensions. A systolic array is one of the most efficient hardware implementations for this workload. The underlying requirement is to convert a convolution operation into a matrix-matrix multiplication where systolic arrays can exploit the data-reuse nature of such multiplication. A systolic array replaces a single processor with a tightly coupled processing elements network. The significant benefit of using a systolic array is reducing direct memory access via a carefully planned data flow, encouraging the exploitation of neighboring data reuse. Previous works have implemented CNN accelerators using systolic arrays with their clever tweaks targeting different performance metrics, including but not limited to latency and throughput [65, 12, 56, 40, 46].

The first motivation of this work is to consider the FPGA architecture when providing the corresponding systolic array implementation. To fully utilize an FPGA device, it is essential to understand the underlying components in the fabric and carefully craft the hardware implementation with that knowledge. We target the Xilinx Ultrascale+ platform, where we apply double data packing and floor-planning according to the specification of DSP48E2 hard blocks available on the device.

The second motivation of this work is to explore how to close the utilization gap while maxing out the throughput bandwidth available in each of those DSP blocks. As FPGA manufacturers produce larger and larger chips targeting data centers, it is crucial to make sure that we can efficiently utilize all the resources available on the chip. It is not easy since different neural networks have different topologies, filters dimensions, and workload distribution. Relying on a one-size-fits-all architecture will inevitably cause a mismatch between computation power supply and demand, resulting in under-utilization.

This work's last motivation is to develop a standard workflow to achieve a throughput focus FPGA-optimized multi systolic array implementation across different neural network topologies. To achieve this, we must determine how to distribute the workload from different layers of computations across all resources of the FPGA chip. One must also consider the effect of systolic array placement given the irregular DSP layout of the underlying FPGA fabric to achieve high-frequency performance. We abstract the FPGA hardware and neural network topology considerations in our optimization flow.

One may naively reason to build one large systolic array to implement the computation and map each layer to that array. This results in severe under-utilization due to mismatches between array size and layer compute requirements [49]. On top of that, in the latest Ultrascale+ platform from Xilinx, some of the larger FPGAs have multiple logic regions connected via a slow interconnection between each region [61].

Supertile [56], a state-of-the-art CNN accelerator, provides an architecture with three processors for convolution layers and one dedicated to fully-connected layers. Their design achieves a very high clock rate at 720 MHz with multi-stage processing to balance the throughput and latency. However, they carefully selected the parameters configuring their design without an automatic workflow. This limits their work on CNN with a nested structure similar to GoogleNet. Researchers have proposed new networks with new architecture components and different design philosophies in the ever-changing AI field. It is not feasible to keep up when a sinking time cost is needed to adjust the parameters for an efficient implementation carefully.

AutoSA [54], a state-of-the-art polyhedral compiler for systolic array, provides an end-to-end flow compilation framework for generating systolic arrays on FPGA, including I/O control. Their result presents designs operating at 250 MHz on the Xilinx Alveo U250 platform. However, the DSP blocks in that platform can operate at a peak frequency of 775 MHz, meaning the throughput bandwidth of these DSPs is not fully utilized.

Our key insight is to formulate physical DSP constraints as a 2D knapsack problem to determine the final systolic array placement. We perform array sizing using a binary search and maximal rectangles algorithm [29] [48], a polynomial-time 2D knapsack algorithm, to find the greatest number of systolic array partitions that can fit on the chip. Our workflow uses these array dimensions and their identified placement from the maximal rectangles algorithm to determine the maximum clock frequency of each partition. The optimization flow generates a solution that minimizes the latency of the slowest partition, hence maximizing overall throughput.

Another critical insight is to treat each SLR as its unique grid and limit the multi-systolic array design within each SLR region. We can then replicate the same design onto each SLR, allowing a linear gain in throughput performance. This allows us to explore more design options in the optimization flow as we further split each SLR into smaller replicating regions. We can also avoid the slow SLR interconnection within one design, negatively impacting the final frequency performance.

We adopt an approach where layers are grouped and mapped to independent systolic arrays implemented on the same FPGA. We develop a systolic array implementation with simple AXI support for I/O using Verilog. We develop a frequency-aware workflow that partitions the FPGA into multiple square systolic arrays and assigns adjacent layers to these partitions with evolutionary techniques.

Summary

1. We create an automatic workflow to synthesize, place and route systolic arrays of different sizes in every staggering location on both the Xilinx VU9P and VU37P Ultrascale+ platforms.
2. We introduce an optimization formulation for generating square systolic arrays and assigning neural network layers into partitions.
3. We use a flexible systolic array RTL generator to create a placement-aware frequency model for the optimization flow. These generated systolic arrays operate between approximately 630–820 MHz,
4. We introduce a workflow combining an evolutionary algorithm, binary search, and a 2D rectangular packing algorithm to create a frequency- and placement-aware optimization flow. Partitioned solutions designed by this workflow operate between 550–670 MHz.

Chapter 2

Background

2.1 Convolution Neural Networks

Convolution neural networks (CNN) are among the most common deep neural network architectures in the machine-learning landscape. Extensive exploration in this type of architecture allows CNNs to become the gold standard of image classification tasks since winning the ImageNet Challenge [31] in 2012. Over the years, researchers have researched applying CNN in different fields, including speech recognition [66, 5], sentiment analysis [27, 19], gesture recognition [64, 39], traffic forecasting [33, 38], medical pattern recognition [41, 30, 7] and etc.

Traditionally, a CNN consists of three significant layer types 1) convolution layers, 2) pooling layers, and 3) fully connected layers. In convolution layers, the primary responsibility is feature extraction via multiple small learning kernels. Several parameters dictate their behaviors, such as padding, stride, and kernel sizes. A few hundred kernels in each convolution layer contribute to the high computation demand. We provide simple pseudocode of a convolution layer in Algorithm 1.

When we calculate the size of the output feature map, we can take the striding and padding parameter into account by following Equation 2.1, where W is the size of the input feature map, F is the size of the filter, P is the number of padding, S is the number of strides, and K is the number of kernels. If the result from Equation 2.1 is not an integer, it indicates that the parameters are invalid.

$$K * \frac{W - F + 2P}{S + 1} \tag{2.1}$$

Algorithm 1: A pseudocode for simple 2D convolution operation

```
Data: Input feature map $I(x_i, y_i)$, Kernel $K(x_k, y_k)$; assuming the $K(0,0)$ locate
      at the middle of the kernel where the height and width are $(h_k, w_k)$
Result: Output feature map $O(x_o, y_o)$
for $y_i = 0$ to $max(y_i)$ do
    for $x_i = 0$ to $max(x_i)$ do
        $sum = 0$;
        for $i = -h_k$ to $h_k$ do
            for $j = -w_k$ to $w_k$ do
                $sum = sum + K(j, i) * I(x - j, y - i)$;
            end
        end
        $O(x, y) = sum$;
    end
end
```

As shown in Algorithm 1, the underlying calculation processes are multiplication and accumulative addition of the results from the multiplication. One common trick is transforming the entire convolution layer into one matrix multiplication for efficient hardware implementation. Given an Input feature map of $[20 * 20]$ with three channels and 64 $[3 * 3]$ filters, we can perform such transformation by following these steps:

1. We stretch every local block of pixels $[3*3*3]$ into a column vector of size $3*3*3 = 27$.
2. We then repeat this process for $(20 - 3 + 1)^2 = 324$ locations along the width and height and pack all these 324 column vectors to form matrix M_I of size $[27 * 324]$.
3. We stretch each filter into a row vector of size $3 * 3 * 3 = 27$ and combine all 64 vectors into to form matrix M_K of size $[64 * 27]$.
4. We can now find the convolution result by multiplying M_K and M_I, forming the resulting matrix M_O of size $[27 * 324]$.
5. The result can then be reshaped back to the proper dimension $[18 * 18 * 64]$.

This is a typical implementation pattern for a forward passing convolution layer.

Pooling layers are similar to the convolution layer, but each filter performs a specific function like max-pooling or average pooling. Their principal responsibility is to reduce dimensionality and exploit the locality of the pixel pool. We provide a simple pseudocode representation in Algorithm 2.

Algorithm 2: A pseudocode for pooling layers

```
**Data:** Input feature map - $I(x_i, y_i)$, Pooling Kernel $K(x_k, y_k)$
**Result:** Output feature map - $O(x_o, y_o)$
for $y_i = 0$ to $max(y_i)$ do
    for $x_i = 0$ to $max(x_i)$ do
        if $AveragePooling$ then
            $acc = 0$;
            for $i = -h_k$ to $h_k$ do
                for $j = -w_k$ to $w_k$ do
                    $acc = acc + I(x - j, y - i)$;
                end
            end
            $O(x, y) = acc/(h_k * w_k)$;
        end
        if $MaxPooling$ then
            for $i = -h_k$ to $h_k$ do
                for $j = -w_k$ to $w_k$ do
                    $maxValue = max(max, I(x - j, y - i))$;
                end
            end
            $O(x, y) = maxValue$;
        end
    end
end
```

Finally, the fully connected layers are the brute force layer in a CNN. They are similar to the output layer of the *multilayer perceptron* (MLP). Effectively, the operation here is matrix-vector multiplication. In this layer, the neurons apply a linear transformation to the input vector using a weights matrix. The output vector $O(x)$ equals to the dot product of the weight matrix w and input vector x where the dimension of x is N plus the bias term w_0 as shown in Equation 2.2

$$O(x) = \sum_{i=1}^{N} (wx_i + w_0) \tag{2.2}$$

This simple dot product operation allows the fully connected layers to aggregate all the information for feature extraction and generate the final classification result.

2.1.1 CNNs benchmark

Different CNNs come with different topologies, and we introduce the CNNs we include in the CNN topology dataset. *AlexNet* [31] is the first CNN that implements the Rectified Linear Units (ReLUs) as their activation functions and contains dropout layers. *AlexNet* contains five convolution layers and three fully connected layers. *GoogleNet* [52] introduces the inception module that reduces the number of tunable parameters from Alexnet's 60 million to 5 million without sacrificing accuracy with even deeper network architecture, as illustrated in Figure 2.1. However, the accuracy improvement diminishes, and the difficulty of training rapidly increases as a neural network increases its depth. *ResNet-50* [25] addresses this issue by adding the concept of residual learning into the architecture by adding shortcut connections.

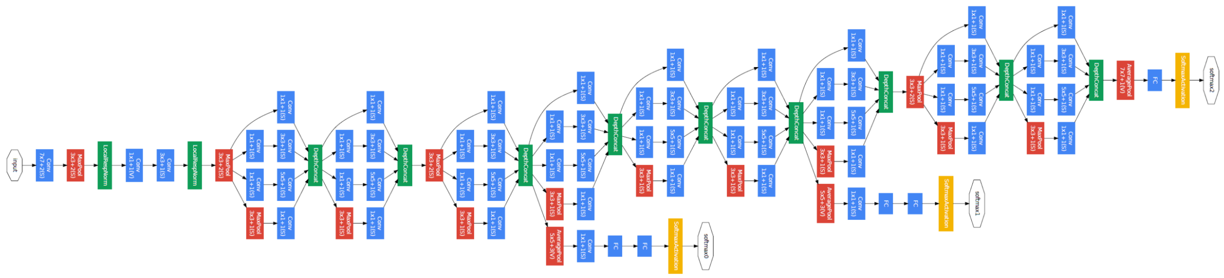


Figure 2.1: An overview of the entire GoogleNet V1 [52]

There are also other CNNs proposed with a focus on their technical aspect, for example, object detection. Ross Girshick et al. [21] developed the *Regions-with-CNN-features* (R-CNN) system introducing the use of a selective search algorithm [53] in a region proposal to bypass the difficulty of selecting the classifying regions. Due to the long execution time of the selective search method, Ross Girshick also proposed the *Fast R-CNN* [20], which uses CNN the generate a feature map before using the same selective search algorithm.

The latest update on *R-CNN* is *Faster R-CNN* from Shaoqing Ren et al.'s work [45]. It achieves near real-time performance by eliminating the selective search and letting the network learn the region proposals. *You Only Look Once* (YOLO) [44] is another example that focuses on the speed of object detection where it uses a single convolution network to predict both the bounding boxes and the classification of those boxes from the complete image instead of multiple individual regions.

MobileNets from Andrew et al. [26] focus on enhancing the efficiency of CNNs for edge devices. This class of efficient models achieves very similar accuracy compared to popular networks like GoogleNet with fewer parameters.

2.2 Systolic Arrays

A systolic array is a type of hardware architecture where homogeneous computation elements, referred to as "units", are connected in a regular geometric fashion with pipelined computation [34]. Figure 2.2 presents a 4×4 systolic array connecting all the processing element (PE) units squarely. These architectures are highly efficient as they extensively exploit data reuse. Due to their efficiency in convolution and matrix multiplication operations, systolic arrays have become the building blocks of neural network accelerators in systems such as Microsoft Brainwave [15] and Google's TPU [3].

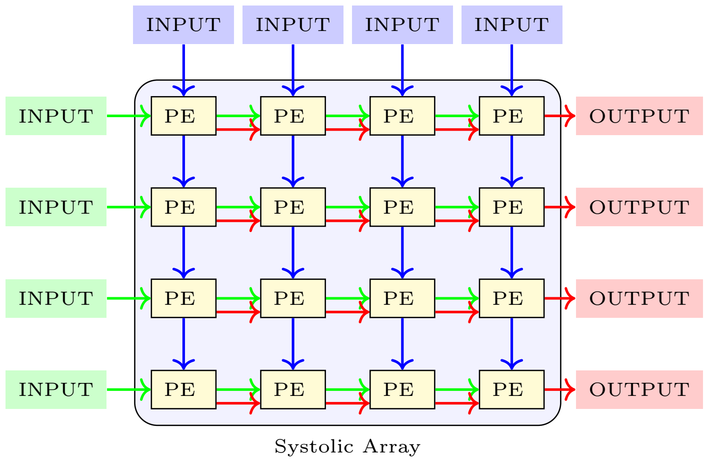


Figure 2.2: A high level overview a traditional 4×4 systolic array.

The performance of a systolic array depends on the underlying hardware architecture and dimensions. In this work, the systolic arrays closely follow a conventional systolic array design: processing elements take in two multiplier inputs (a, b) and an init (or flush) signal and are composed of a multiply-accumulate unit and a drain pipeline with a valid bit.

Under-utilization vs. Performance Gain

A fundamental limitation of mapping CNNs to systolic arrays is the threat of low array utilization. Since the computation workload in each layer is different, a one-size-fits-all systolic array will inevitably cause under-utilization. In Figure 2.3, we plot the percentage of the idle array and the cycle count needed against the size of the systolic array for the first convolution layer of GoogleNet V1 using SCALESim [47]. SCALESim [47] is a CNN accelerator simulator that provides a cycle-accurate timing based on a systolic array architecture.

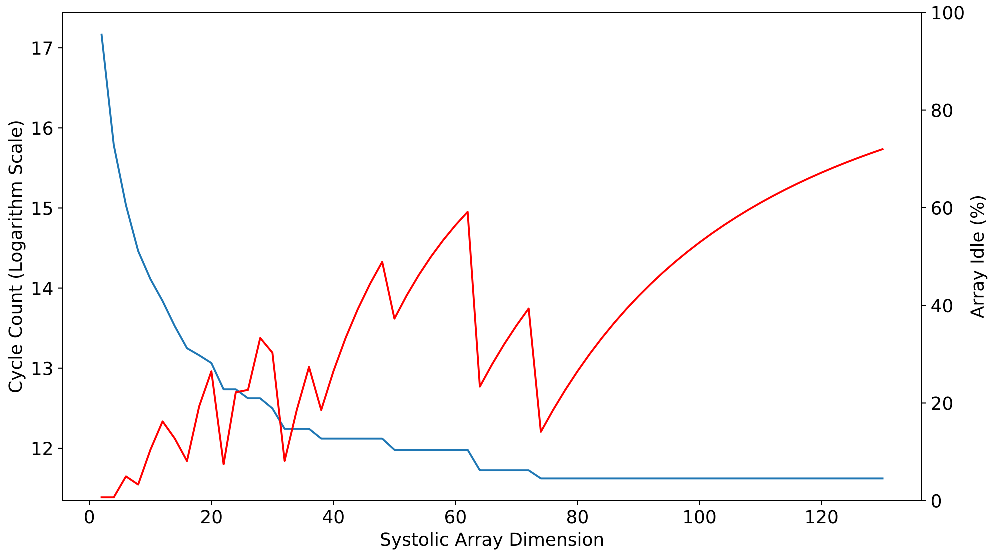


Figure 2.3: Under-utilization percentage and cycle count against systolic array size.

As we can see, more arrays stay idle as the systolic array dimension increase, whereas the gain in cycle count becomes minimal. SCALESim [47] simulates the controller for feeding data into the systolic array in the simulation. Due to the implementation of the controller having a coarser update step-size, SCALESim use the same controller to feed a set of systolic arrays with different dimensions result in a mismatch. The unusual spikes in Figure 2.3 come from this quantization effect of SCALESim.

2.3 Xilinx Ultrascale Platform

We target the VU37P and VU9P chips on the Xilinx UltraScale platform. Some features specific to this platform allow us to realize a high-performance systolic array design on these chips.

Super Logic Regions (SLRs)

To accommodate the rapidly increasing capacity of Look-up Tables (LUTs), registers, Random Access Memory (RAM), and Digital Signal Processing (DSP) Blocks, manufacturers need to produce larger dies. Xilinx proposed their *Large FPGA Methodology* back in 2012, suggesting the architecture that mounts multiple SLR components on a passive silicon interposer where the SLR components connect to its adjacent SLR components via high-bandwidth, low latency connections called *Stacked Silicon Interconnect* (SSI)[61]. On VU37P and VU9P, there are three SLRs on both of the dies. Each SLR region contains multiple clock regions where the LUTs, RAMs, and DSP slices reside in a column arrangement. A device view of the die is provided in Figure 2.4.

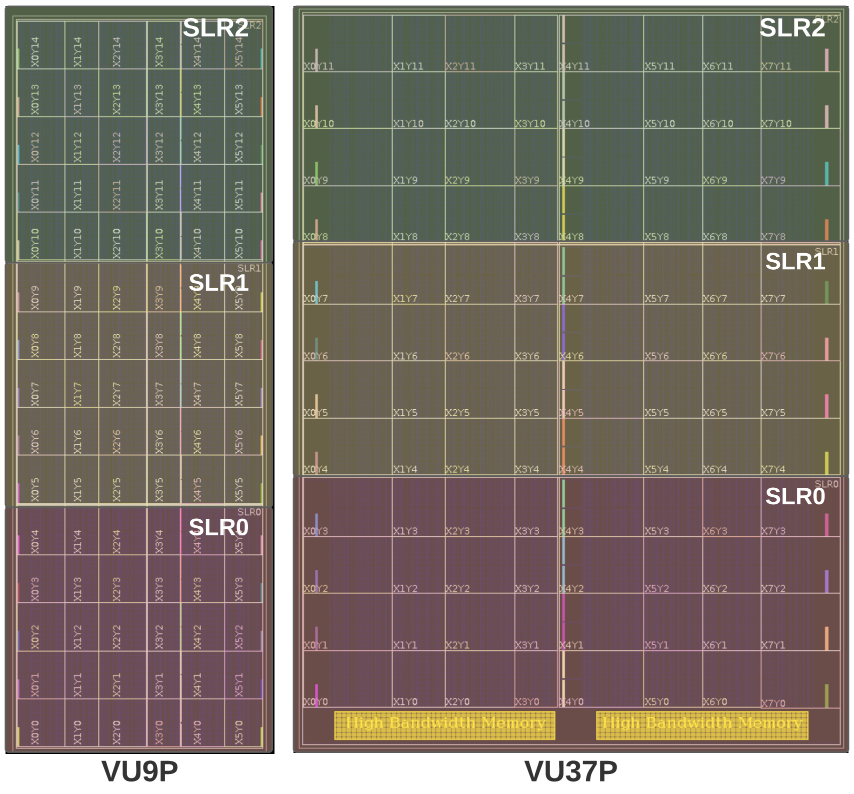


Figure 2.4: Device view of VU37P and VU9P

UltraScale DSP resource

We construct the systolic array by utilizing the DSP slices on the FPGA. There are 6840 and 9024 DSP slices on VU37P and VU9P, respectively [57]. The DSP slices we are targeting are *DSP48E2*, an update of *DSP48E1* from the 7-series FPGA. Some important improvements include a wider multiplier with 27×18 and a wider pre-adder width of 27 bits. On top of improvements in the DSP computation capability, DSP resources are organized as a DSP tile on the device [58]. Each tile contains one 36K Block RAM, five configurable logic blocks (CLBs), and two *DSP48E2* slices, as illustrated in Figure 2.5.

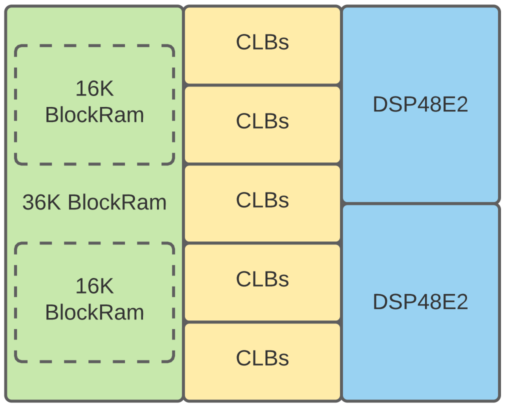


Figure 2.5: A structured view of a DSP tile on Xilinx UltraScale Platform

For VU37P, each SLR contains 24 clock regions with eight columns and four rows of DSP slices. In each SLR region, 32 columns of DSPs are spread across all clock regions in an asymmetry fashion. Each column in each SLR region has 94 rows meaning that each SLR region contains $32 \times 94 = 3008$ DSP blocks. For VU9P, each SLR contains 30 clock regions with six columns and five rows of DSP slices. There are 19 columns of DSPs in each SLR region spread across all clock regions. Each column in each SLR region has 120 rows meaning that each SLR region contains $19 \times 120 = 2280$ DSP blocks.

UltraScale memory resource

We utilize the on-chip memory to supply the data flowing into the systolic array. Instead of relying on the distributed RAM, we mainly depend on the block RAM hard blocks due to the better local connectivity, as shown in Figure 2.5. The specific hardblock we infer is *RAMB36E2* which is also an update from their 7-series variant. These BRAMs consist of a 36Kbit storage area and two independent access ports [59]. We can treat these BRAMs as two independent 16Kb BRAM.

Other than the readily available BRAMs, the target device also provides UltraRAM and HBM DRAM in applications requiring more on-chip memory. UltraRAM is also compatible with the physical columnar architecture on the FPGA dies. Each clock region contains one column of 16 rows of UltraRAM blocks. UltraRAM is another flexible, high-density 288Kb memory block meaning that each UltraRAM block has $8\times$ the capacity of a BRAM. However, the UltraRAM blocks have only one clock input and only support reading or writing per port per cycle.

Besides the memory embedded in the columnar architecture, VU37P also provides another memory stack - High Bandwidth Memory (HBM). Its physical location on the die was shown at the bottom of the device view in Figure 2.4. These memories can only be accessed via the 16×16 AXI crossbar and require a specific AXI High Bandwidth Memory Controller for data transactions. We have summaries of the amount of on-chip memory available on VU37P and VU9P in Table 2.1

Table 2.1: Memory Resource Summary on VU37 and VU9P

	VU37P	VU9P
BRAM (Mb)	70.9	75.9
UltraRAM (Mb)	270.0	270.0
HBM (GB)	8	-

2.4 Evolutionary algorithm

An *Evolutionary algorithm* [55, 22] is a class of algorithms that generates a set of individual candidates in every evolution to evaluate the problem space. These algorithms take inspiration from biological evolution with the idea of *survival of the fittest*. On an abstract level, the algorithm uses an objective function (fitness function) to evaluate the performance of candidates in each evolution. This type of algorithm selects top-performing candidates via the *deterministicSurvivorSelection* procedure and evolves the distribution representative of the solution space. The algorithm then produces the next generation of candidates with better fitness. This process repeats until future generations stop improving or the algorithm meets a user-defined termination condition.

This technique has been shown to solve complex *Black Box* optimization problems. The actual implementation is not bound to a specific agent or environment. The significant difference between different implementations depends on the *Adaptation Policy*, which determines how to pick the next generation, and the mathematical approach to estimate the expected reward for each generation. The *Adaptation Policy* affects the final quality of the solution by determining how to refine candidate solutions throughout the series of evolution steps. A set of mutations must be performed in each step to produce an ensemble of potential solution models. *Evolution Strategy* is effective for applications where the computation of gradients is intractable, allowing it to be a more robust solution compared to the traditional *Re-enforcement Learning* with back-propagation. In *Covariance*

Matrix Adaptation Evolution Strategy (CMA-ES), the algorithm can discover the problem structure by representing the candidate solution as a distribution of random variables.

2.4.1 Covariance Matrix Adaptation Evolution Strategy

We also experiment on two other optimization techniques - *Simple Genetic Algorithm* and *Hyperopt* before picking CMA-ES.

Simple Genetic Algorithm starts with an initial population pool where a global cost function is used to evaluate each individual. The algorithms then pick the portion of the population with the highest fitness score to create a mating pool where we perform a crossover operation between each set of the mating population to create a new generation of offspring. We then apply random mutation in the offspring and repeat the process until reaching a user-defined ending condition.

Hyperopt [8] is a library that provides parallelization infrastructure and three specific algorithms implementation, including 1) random search, 2) Tree of Parzen Estimators (TPE), and 3) Adaptive TPE to conduct Sequential model-based Bayesian optimization for hyperparameter optimization.

Our previous work [12] reveals that CMA-ES produces the best quality of results compared to these two techniques when adapted into a similar type of optimization problem.

In a simple *Evolution Strategy*, we sample the current generation of candidates from a normal distribution with a mean μ_i and a fixed standard deviation σ. After evaluating the fitness of the entire population, we set μ_{i+1} to be the best solution in the current population. We then sampled the next generation of candidates from this new normal distribution with μ_{i+1} and σ.

The most significant difference of CMA-ES compared to the simple *Evolution Strategy* is that its *Adaptation Policy* allows adjustment to the search space for the next generation. This clever insight enables the algorithm to spread out and cover more search space when compared to the *Simple Genetic Algorithm*, which has a static noise and spread parameter across generations. CMA-ES samples the candidates from a multivariate normal distribution and its formal description is provided in Equation 2.3 [23].

$$x_{i+1}^{k} \sim \mu_i + \sigma_i \mathcal{N}(0, \mathcal{C}_i) \text{ for } k \in 1, ..., \lambda \tag{2.3}$$

In Equation 2.3, x_{i+1}^{k} is the k-th candidate in generation $i+1$; μ_i is the mean of the search distribution at generation i; σ_i is the standard deviation of the search distribution at generation i; σ_i is the standard deviation of the search distribution at generation i and $\mathcal{C}_i$ is the covariance matrix of the search distribution at generation i. $\mathcal{N}(0, \mathcal{C}_i)$ is the multivariate normal distribution with zero mean and the covariance matrix $\mathcal{C}_i$. At last, λ is the population size of each generation.

CMA-ES iteratively samples new generations of candidates via updating μ, σ, and $\mathcal{C}$ based on the population's fitness in the last generation. Instead of calculating the exact $\mathcal{C}$, CMA-ES use the maximum likelihood estimate of the covariance matrix $\mathcal{C}$. For a set of N two-dimensional random samplings (x, y), we estimate the two-dimensional $\mathcal{C}$ by calculating the following terms:

$$\mu_x = \frac{1}{N} \sum_{j=1}^{N} x_j \tag{2.4}$$

$$\mu_y = \frac{1}{N} \sum_{j=1}^{N} y_j \tag{2.5}$$

$$\sigma_x = \frac{1}{N} \sum_{j=1}^{N} (x_j - \mu_x)^2 \tag{2.6}$$

$$\sigma_y = \frac{1}{N} \sum_{j=1}^{N} (y_j - \mu_y)^2 \tag{2.7}$$

$$\sigma_{xy} = \frac{1}{N} \sum_{j=1}^{N} (x_j - \mu_x)(y_j - \mu_y) \tag{2.8}$$

However, the five terms $\mu_x, \mu_y, \sigma_x, \sigma_y, \sigma_{xy}$ only provide us the estimate of two-dimensional covariance matrix in the current generation. To adjust and move the search space across the 2D plane, CMA-ES tweaks the covariance matrix update. CMA-ES achieves this by estimating the covariance matrix of the next generation by using a certain amount of best sampling from the current generation. It first calculates the x and y mean of the N_{best} samplings in the current generation.

$$mean_x = \frac{1}{N_{best}} \sum_{j=1}^{N_{best}} x_j \tag{2.9}$$

$$mean_y = \frac{1}{N_{best}} \sum_{j=1}^{N_{best}} y_j \tag{2.10}$$

To estimate the 2-dimensional covariance matrix for the next generation, we need three covariance terms: σ_x, σ_y and σ_{xy}.

$$\sigma_x = \frac{1}{N_{best}} \sum_{j=1}^{N_{best}} (x_j - mean_x)^2 \tag{2.11}$$

$$\sigma_y = \frac{1}{N_{best}} \sum_{j=1}^{N_{best}} (y_j - mean_y)^2 \tag{2.12}$$

$$\sigma_{xy} = \frac{1}{N_{best}} \sum_{j=1}^{N_{best}} (x_j - mean_x)(y_j - mean_y) \tag{2.13}$$

The clever trick here is to calculate the average value of the selective N_{best} samplings in Equations 2.9 and 2.10 instead of using the average value over the entire population. This trick allows the algorithm to adjust the search space by estimating the covariance matrix of the next generation. Suppose we use the average value of the entire current generation. In that case, the result is just an estimation of the covariance matrix of the current generation, which does not help the algorithm adjust its search space. Finally, CMA-ES samples a new set of candidates from this new covariance matrix constructed with the updated mean $mean_x$, $mean_y$ and σ_x, σ_y, σ_{xy}.

We apply CMA-ES on a shifted second-order Schaffer function, a standard testing function for black-box optimization [50, 14, 22], to visually demonstrate how a two-dimensional CMA-ES evolve across its early generations in Figure 2.6. Equation 2.14 defines a second-order Schaffer function where its global minimum locates at $(0, 0)$.

$$f(x, y) = 0.5 + \frac{\sin^2(x^2 - y^2) - 0.5}{[1 + 0.001(x^2 - y^2)]^2} \tag{2.14}$$

We represent the best candidate of the current generation as a red dot and the best candidate of the last generation as a green dot. All the other samplings in that generation are represented as transparent blue dots. The global optimal in Figure 2.6 resides in the brightest spot at the bottom left corner - $(-2, -2)$.

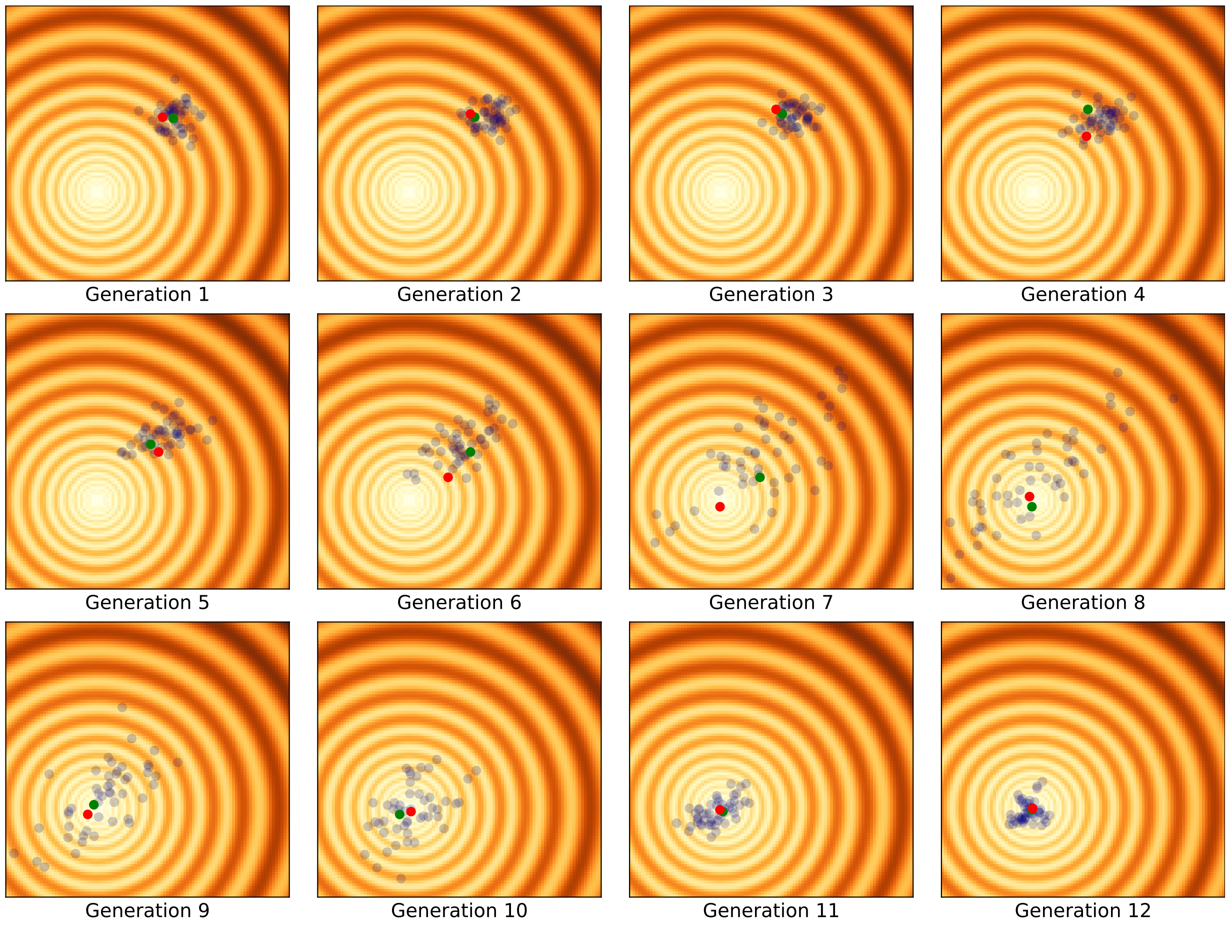


Figure 2.6: A visual presentation of CMA-ES candidates per generation

In Figure 2.6, the algorithm expands its search space starting at generation 3. By generation 9, the best candidates had already reached proximity to the global optimal.

We borrow an open-source python implementation of CMA-ES [24].

2.5 KnapSack Optimization Algorithm

The 2D knapsack is a family of problems in combinatorial optimization [1]. The general definition of the knapsack problem is "Given n items, each of a given size, and some bins of a certain capacity. Determine an assignment from item to bins using as few bins as possible." [1, 10] When applied to a 2D space, this rectangular space is referred to as the "knapsack". The problem has been proven to be NP-hard [10]; a brute force approach with $O(4^{c^2})$ runtime complexity [35] is the only way to find the optimal solution.

2.5.1 Maximal Rectangles Algorithm

The maximal rectangles algorithm [29] is a polynomial-time $O(n^3)$ greedy algorithm for solving the 2D knapsack problem. This algorithm is an improvement from the *Guillotine algorithm*, another rectangular bin packing algorithm based on the *Guillotine split placement*. The *Guillotine split placement* places the rectangles to the corner of a free knapsack and then splits the remaining L-shaped space into two disjoint free rectangles via either a horizontal or a vertical cut, as illustrated in Figure 2.7.

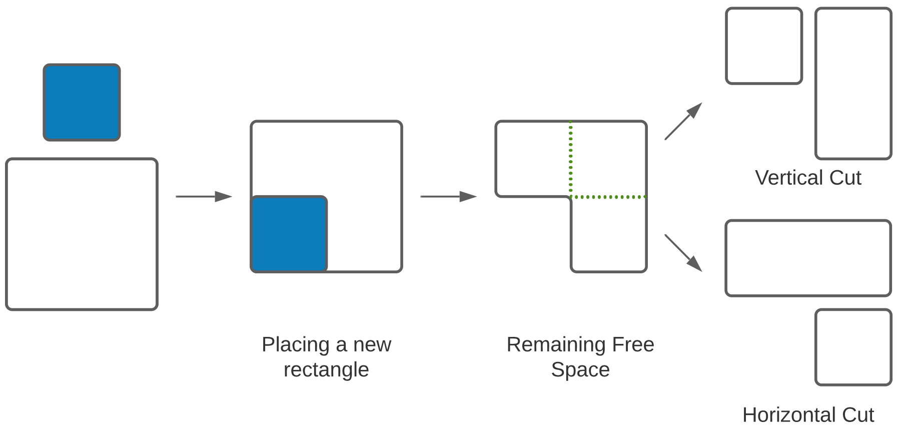


Figure 2.7: A visual presentation of guillotine split placement

The *Guillotine algorithm* starts with one new rectangular bin. It then performs the *Guillotine split placement* and creates a list of pairwise disjoint rectangles representing the remaining free space of such bin. It then iteratively repeats the cutting and replaces the original rectangular area with the two smaller rectangular spaces. This procedure then continues until none of the spaces in the list can fit the next item. The algorithm then works on the next bin if there is any.

The maximal rectangles algorithm is very similar to the *Guillotine algorithm*. However, it tweaks the *Guillotine split placement* procedure by considering both possible splits and adding both the rectangular space into the list. The variation of the maximal rectangles algorithm will always pack the largest item in the queue in each iteration. The algorithm also ends when there are no more items to pack or more room in the bin.

In our optimizer, we adapt an open-source python implementation of the KnapSack algorithm called *RectPack* library [48].

Chapter 3

Our Proposal

3.1 Overall Architecture

This section proposes a throughput-optimized FPGA-based CNN accelerator using systolic array partitions. Next, we introduce splitting the FPGA into smaller replicating regions for further design exploration.

3.1.1 Multi Systolic Array Architecture

We present an overview of the accelerator's architecture in Figure 3.1. The architecture consists of multiple square systolic arrays where each systolic array partition operates at its optimal frequency with its separate clock input. Larger systolic array partitions usually operate at a lower frequency to avoid timing violations. If all the partitions follow the same clock input, the smaller partitions are not operating at their highest possible frequency. We, therefore, separate each partition's clock input, allowing all partitions to operate at their optimal frequency to improve the overall throughput performance. We first split the workload of a CNN into several groups continuously, meaning that each group will contain consecutive layers. We then assign each workload grouping to one systolic array partition. The whole architecture then processes the CNN in a pipelined fashion. We decide on this splitting method for these significant reasons 1) ensure high utilization of the FPGA resources, 2) simplify design space for optimization, and 3) allow independent frequency optimization for each array.

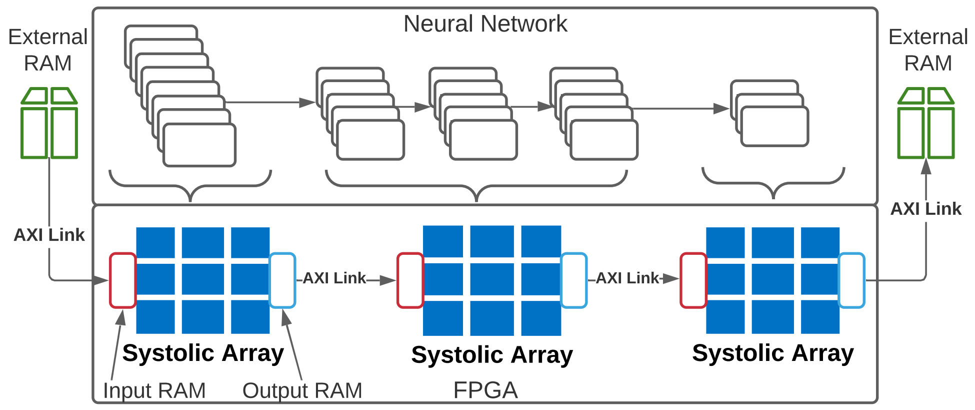


Figure 3.1: A visual presentation of the pipeline in the partitioned architecture.

Each systolic array operates at different frequencies while communicating to its neighboring arrays using the AXI interface. The AXI interfaces will adopt the global clock at 400 MHz syncing across all channels while the computation array uses a separate clock domain operating at around 600-700 MHz. Since the on-chip memory resource are limited, and CNNs with millions of parameters will easily exceed the storage capacity on the FPGA die, we need to stream the pixel and weight inputs from the external RAM to our systolic arrays.

We focus on using square-shaped systolic arrays since most of the CNNs' layers have square-sized kernels. Matching the array and kernel shape allows simpler and direct data movement where we can equally exploit data reuse along the horizontal and vertical axis in the systolic array.

3.1.2 Device Logic Region Splitting Technique

This section describes how we plan to explore designs targeting different regions on different FPGA die. We observe that we can exploit FPGA symmetry across SLRs and replicate designs. Instead of spreading the workload overall 3 SLRs, we configure our optimizer to only spread the workload on one SLR and half of one SLR. For results using one SLR, we convert the result to a full-chip design by replicating the smaller design three times, yielding the 3× replicating designs. We also convert the result to a full-chip design using half of the SLR by replicating the design six times, yielding the 6× replicating designs. We explore all of these design spaces by changing the configuration file of the optimizer.

To realize this replication concept in the optimizer, we create different *KnapSack profiles* for VU37P and VU9P. `xcvu37p-full` is the basic design with no replication and spreading partitions onto the full FPGA die. `xcvu37p-3-times` produce the 3× replicating design focusing on putting all partitions in one SLR region. Both `xcvu37p-6-times-x` and `xcvu37p-6-times-y` produce the 6× replicating design by splitting each SLR region in half along the x-axis and y-axis, respectively.

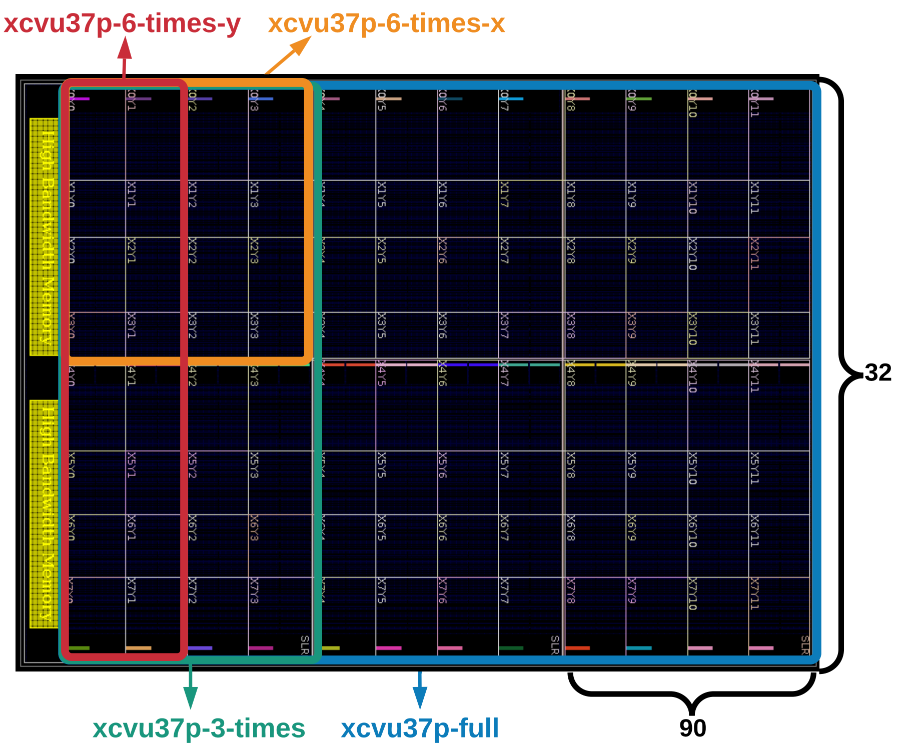


Figure 3.2: Visual representation of Knapsack profile modeling VU37P

However, each region is not symmetrical along the x-axis. Eleven columns are located in clock regions X0 – X2, and eight in clock regions X3 – X5. `xcvu9p-full` and `xcvu9p-3-times` follow similar ideas as `xcvu37p-full` and `xcvu37p-3-times` generating the non-replicating designs and the 3× replicating designs respectively. `xcvu9p-6-times-x-l` restrict the design to the left side of the SLR region, including clock region X0 – X2 with 11 columns of DSPs. `xcvu9p-6-times-x-r` restrict the design to the right side of the SLR region, including clock region X3 – X5 with eight columns of DSPs. `xcvu9p-6-times-y` produce the 6× replicating design by splitting each SLR region half along the y-axis.

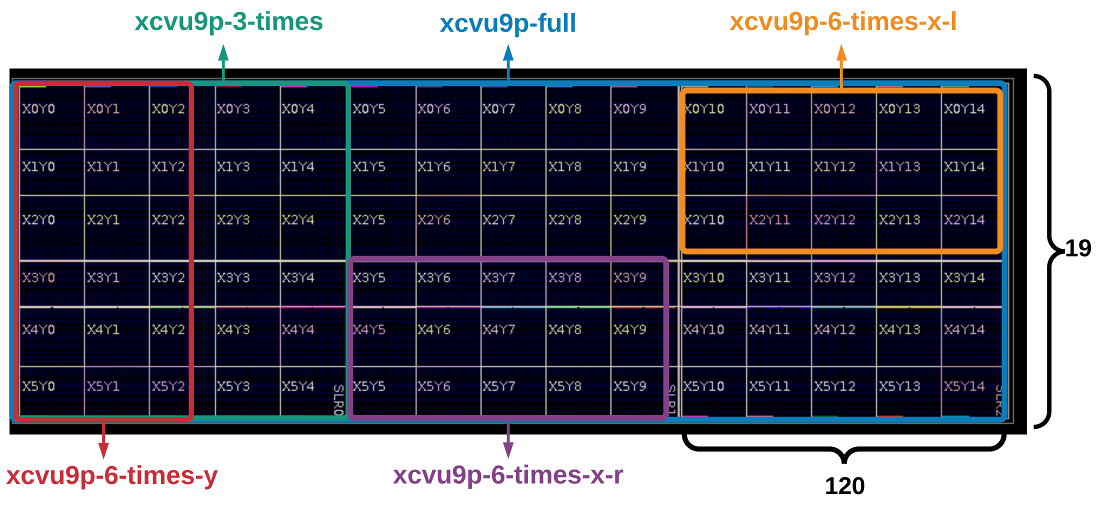


Figure 3.3: Visual representing of Knapsack profile modeling VU9P

The complete configuration files targeting VU37P and VU9P are attached in Appendix D and E with the exact command to recreate the dataset we use for the analysis and result chapter.

3.2 Problem Formulation

This section introduces the challenges we encountered when reaching the proposed architecture on an FPGA. Next, we summarize all the constraints and combine them into one formulation.

3.2.1 Challenges

The proposed architecture has a huge design space and many tunable parameters. To obtain a feasible implementation with optimized throughput performance, we must consider the placement constraints and frequency performance implied by the underlying FPGA fabrics.

Mapping Computations to Partitions

To correctly map a CNN's computation to multiple partitions, we must tackle the problem of 1) splitting neural network layers across the arrays on the chip (workload allocation) and 2) sizing multiple systolic arrays (resource allocation). However, we cannot resolve these two problems separately. The optimal amount of resources allocated to a partition depends on the amount of workload allocated to this partition, while the optimal amount of workload allocated to this partition depends on the resources available. We create a nested loop design with CMA-ES to jointly search for the best resource and workload allocation to tackle this entangling problem.

Placement Constraints

Assuming we now have a set of adequately sized systolic arrays, we still need to place and route each systolic array on the FPGA chip for the final design implementation. Unlike the previous architecture in Chan et al. [12], which relies on a limited 1D shape having a fixed 9×1920 thin rectangular structure, fitting multiple arrays with arbitrary dimensions comes with the problem of "overlapping." When overlapping happens, the two partitions fight for the same resource on the FPGA dies, producing infeasible placement and implementation failure. One may argue that we can use the sum of area to determine whether a placement is possible. However, overlapping can still happen even when the total area sum of each partition is smaller than the area available in the grid, as shown in Figure 3.4.

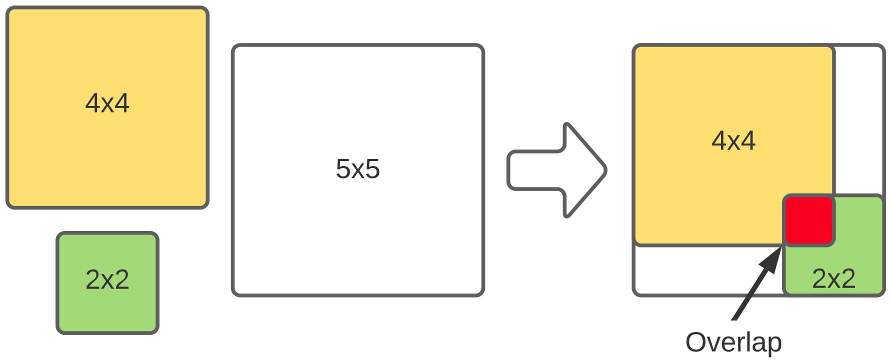


Figure 3.4: A representation when overlapping occurs.

Avoiding overlapping objects is a primary consideration under the 2D knapsack problem formulation. We, therefore, model our DSP placement into a 2D knapsack problem. We model each partition as a unique object and the available DSP grid area as the knapsack. We also constrain each partition to be orthogonal to the edge of the knapsack, i.e., the edges of square partitions must be parallel to the edges of the bounding box. We abstract each SLR on the VU37P as a 64×94 bin and each SLR on VU9P as a 19×120 bin. This unique abstraction mimics the physical arrangement of DSP48E2 blocks on both chips. Thus, the 2D knapsack problem formulation for this case becomes: "Given a set p of square partition side lengths, determine whether all the squares in p can be packed into three $X \times Y$ bins where the X and Y values vary depending on the target chip." While bin packing can introduce DSP wastage, most of our experiments with 100% DSP utilization fail in the routing stage.

Frequency Optimization

On top of fitting all the partitions on the FPGA, the placement of each partition can lead to varying operating frequencies for each systolic array. We demonstrate this property in Figures 3.5 and 3.6, presenting the frequency data when a 2×2 systolic array being placed across the clock region $X3Y0$ to $X5Y4$ of VU9P. We collect the frequency data in the report after placing and routing a 2×2 systolic array in every staggering location with a step size of 1 in both the vertical and horizontal directions. In Figure 3.5, we visualize the impact of different placement on the operating frequency of a systolic array. The yellower region represents a placed design with a higher operating frequency, whereas a block with a red block represents a design with a relatively lower operating frequency.

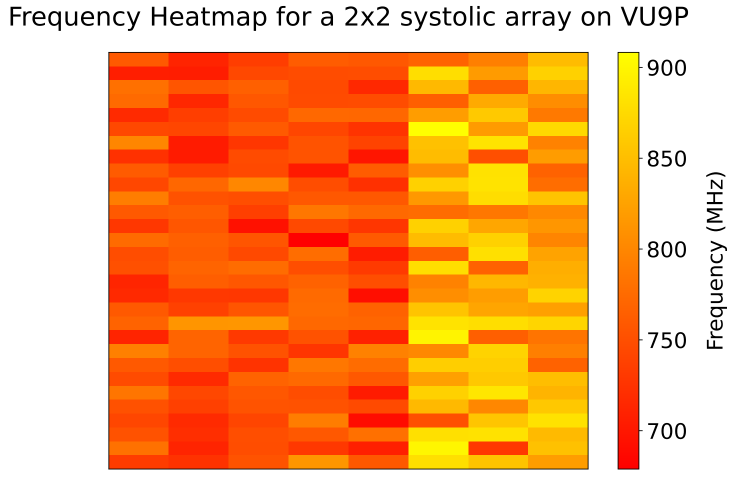


Figure 3.5: A heatmap representation of the operating frequency of a 2×2 systolic array

In Figure 3.6, we visualize the distribution of those frequencies in Figure 3.5. We can see that the frequency can vary from 678 MHz to 908 MHz depending on its placement on the FPGA die, even for a small 2×2 systolic array.

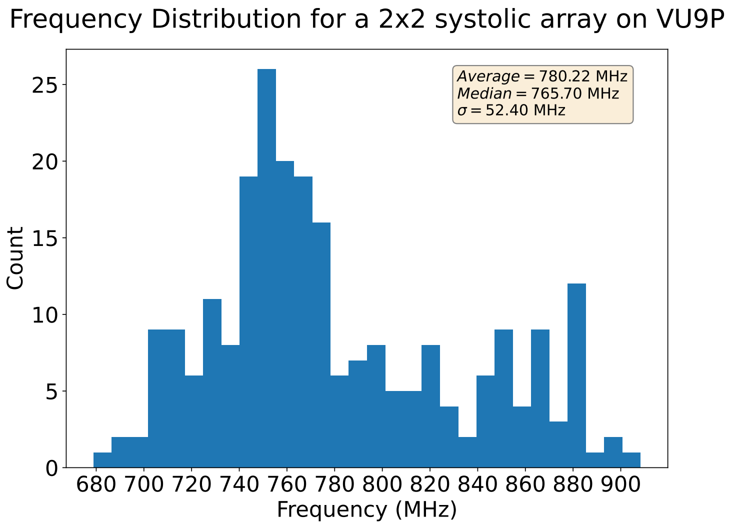


Figure 3.6: A distribution histogram of the operating frequency of a 2×2 systolic array

In Figure 3.7, we can see this trend continues when we collect the frequency data for systolic arrays with larger dimensions. This frequency discrepancy is most likely due to the interleaving column arrangement of different hard blocks controlled by the manufacturer. A shift on the DSP placement constraints also alters the available LUTs, RAMs, and other hard blocks to be used in place and route, causing a different placement and routing result and thus affecting the frequency performance. This 230 MHz frequency gap leaves us considerable room for improvement to determine the best placement for each partition.

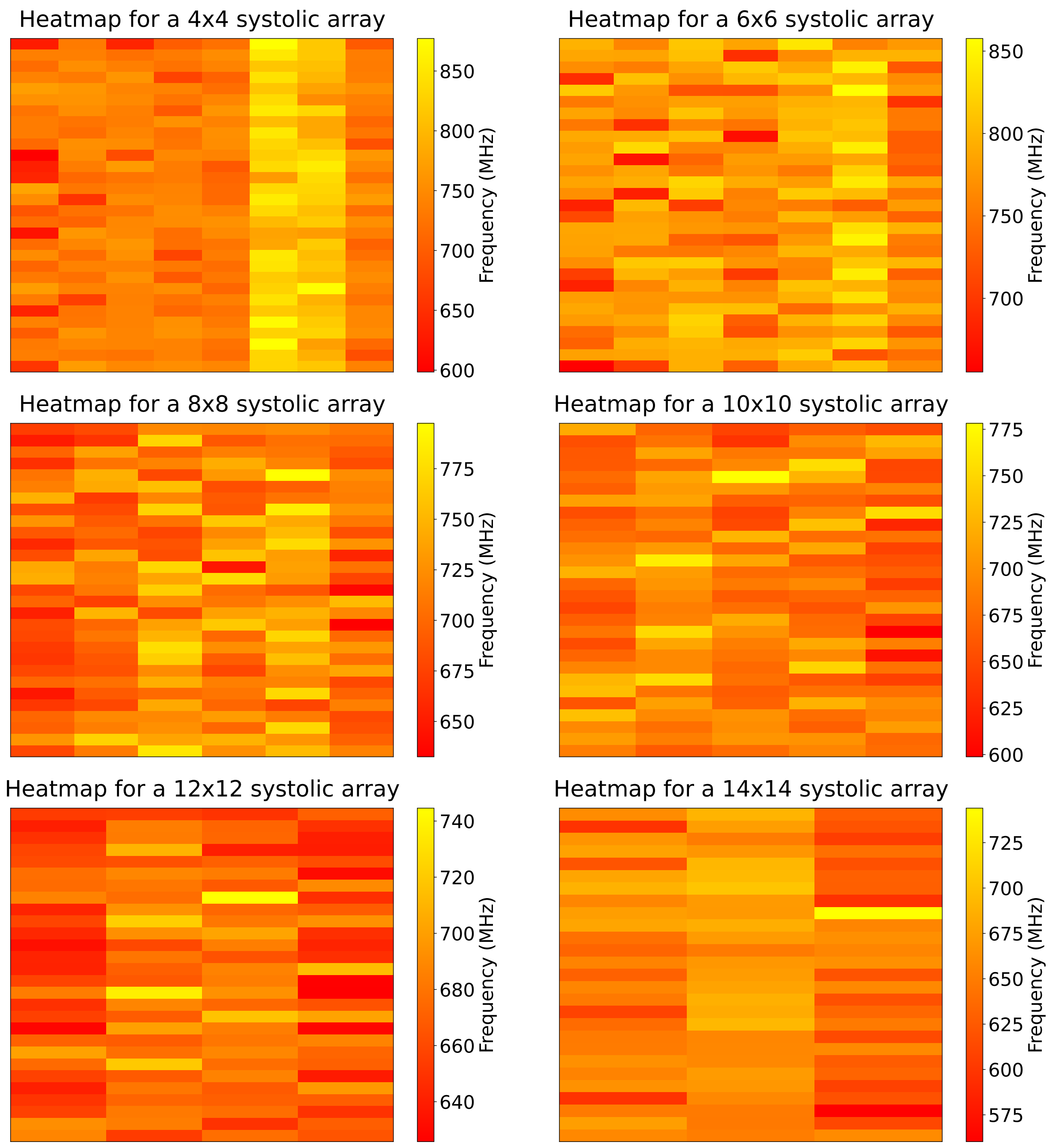


Figure 3.7: A heatmap visualization of frequencies collected covering clock region $X3Y0$ to $X5Y4$ on VU9P for systolic array size from 4×4 up to 14×14.

We present two implementations of the same 8×8 systolic array subjecting to the same area constraints using the same source code. Using the recommended implementation flow suggested in Xilinx documentation, the implementation achieves operating frequency at 482 MHz, while the design using our custom Tcl script can achieve an implementation that operates at 576 MHz. We accomplish this by 1) selectively pre-routing some most critical paths before routing the entire design and 2) using a placement profile to force Vivido to place in a more congested fashion with our specific floorplanning constraints. This is another room for improvement for operating frequency.

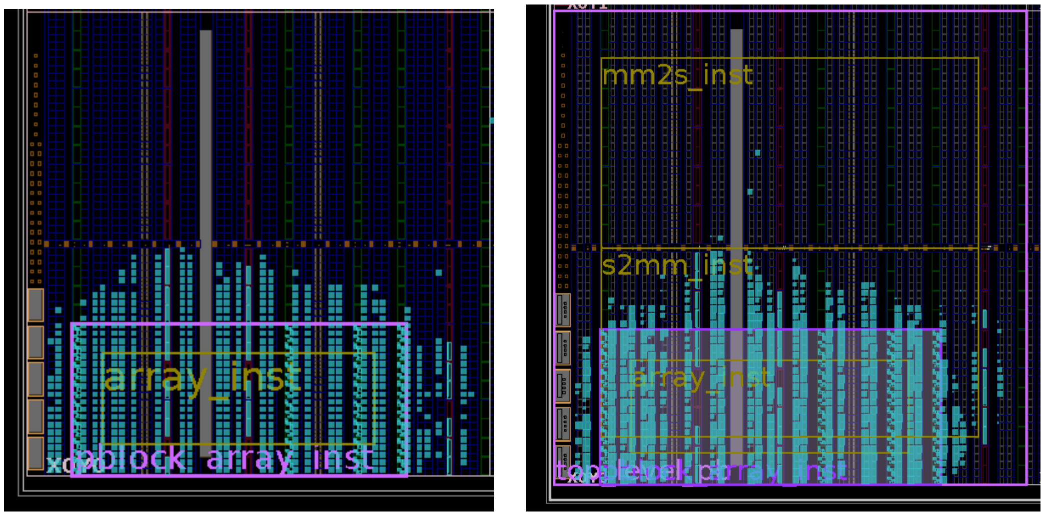


Figure 3.8: A side-by-side device view of the same 8×8 systolic array implemented by Vivado 2018.3 with default non-project flow (left) and own custom TCL flow (right).

To achieve the best frequency for each partition, we obtain a similar set of frequency distribution as we present in Figure 3.5 for every dimension of the systolic array on every possible placement. We then incorporate this data set into our optimization workflow. Since building this data set is tedious and time-consuming, we have developed custom scripts to speed up this one-time process.

3.2.2 Complete Formulation

The proposed architecture has a huge design space and many tunable parameters. To obtain a feasible implementation with optimized throughput performance, we must consider the placement constraints and frequency performance implied by the underlying FPGA fabrics.

We provide a complete formulation, as shown in Equations 3.1 to 3.6, by combining 1) the overall architecture design, 2) the "packability" constraint, and 3) frequency optimization concern.

$$\min_{l,p}\left(\max_{x}\left(\sum_{y\in l[x]}\frac{cyc[y][p[x]]}{freq[p[x]][c[x]]}\right)\right)\text{, subject to:} \tag{3.1}$$

$$\forall x, p[x]\geq 2 \;\&\; p[x]\%2=0 \;\&\; \sum_{x=0}^{K-1}p[x]^2\leq MaxPEUnit \tag{3.2}$$

$$\forall x, l[x]\geq 1 \;\&\; \sum_{x=0}^{K-1}|l[x]|=N \tag{3.3}$$

$$\forall x, 0\leq c_x[x]<c_x[x]+p[x]\leq W \tag{3.4}$$

$$\forall x, 0\leq c_y[x]<c_y[x]+p[x]\leq H \tag{3.5}$$

$$\forall\text{ pairs}(i,j)\in x\text{ where }i\neq j, R(i)\cap R(j)=\emptyset \tag{3.6}$$

We provide the definition of the variables from Equation 3.1 to Equation 3.6 in Table 3.1

Table 3.1: Variable definitions from Equation 3.1 to Equation 3.6

Variable	Definition
x	Index of the partition
$cyc[][]$	The timing model for the systolic array implementing a particular network layer
$freq[][]$	The frequency model for a given systolic array size and location
$p[x]$	The dimension of the square array corresponding to partition x
$l[x]$	The set of layers mapped to the partition x
$c[x]$	The representation of the location assignment containing the xy coordinate $(c_x[x], c_y[x])$ of the bottom-left corners of partition x
$MaxPEUnit$	The maximum available number of PE units available on the either chip. $MaxPEUnit = 13680$ for VU9P and $MaxPEUnit = 17280$ for VU37P
W	The width of the knapsack
H	The the height of the knapsack
$R(x)$	The geometry 2D region covering the partition x where $R(x) = [[c_x[x], c_x[x] + p[x]], [c_y[x], c_y[x] + p[x]]]$

The 2D knapsack problem formulation is used to evaluate whether the constraint in Equations 3.4 to 3.6 is satisfied. Equations 3.4 and 3.5 restrict every partition to be within the boundary of the knapsack. Equations 3.6 prohibit the occurrence of overlapping between any pair of partitions. In the end, the optimizer needs to provide us a design solution containing a list of layers assigned to each partition (l), a list of dimensions for each partition (p), and a list of bounding boxes for each partition (c); in other words, a floorplan for a fully partitioned systolic array design. We can also change the bin dimension and the number of bins in Equations 3.4 and 3.5 to optimize specific clock regions instead of the whole SLR region on the die.

3.3 Workflow

This section provides an overview of our workflow containing two major stages - database generation and optimization. Then, we introduce what kind of data we need for the optimization and how to generate them. Finally, we explain the different variations of optimization we perform with our optimizer.

In Figure 3.9, we provide a high-level overview of the entire workflow. We use custom scripts to generate the required cycle, $cyc[][]$, frequency, $freq[][]$ data and the design checkpoint (DCP), *.dcp* files in stage 1 – *Database Generation*. The data is then used in stage 2 – *Optimization*. We search for a legal and optimized partitioned design in the second stage. We then realize the design by re-using the design checkpoint files from stage 1 to reassemble the design on the target device.

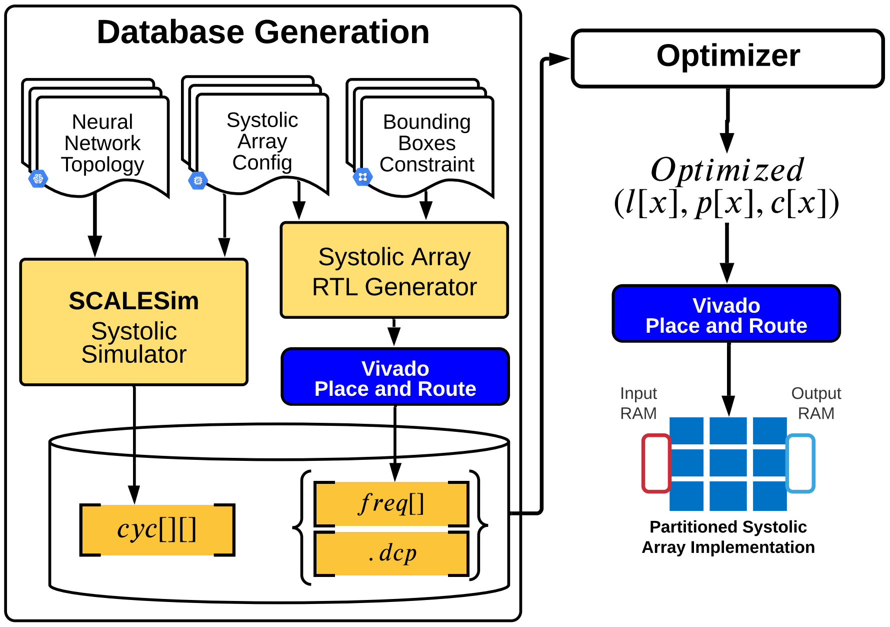


Figure 3.9: High-level diagram of the entire workflow.

3.3.1 Stage 1 – Database Generation

We present an overview of how we use each abstract input to create the database needed by the optimizer in Figure 3.9. We use two different sets of data to construct the database. We use SCALESim to build the cycle model - $cycles[][]$, and the systolic array RTL generator to construct the frequency model - $freq[][]$.

1. **Cycle Modeling**: To solve the optimization problem, we require an efficient method for calculating the number of cycles ($cycle[][]$) needed to compute the outputs for a layer given a systolic array configuration. To avoid doing RTL simulation for every

layer in every network with every possible array, we use SCALESim [47], a convolutional neural network simulator that provides cycle-accurate timing simulations for different accelerator configurations as shown in Figure **??**. We provide SCALESim, the neural network topologies from MLPerf [43] and our configurations of the systolic arrays as input. This yields a pre-computed cycle dataset for every possible combination of layer type (convolution and fully-connected) and square systolic array size. We create custom scripts in Python to interface with SCALESim to automate and speed up the workflow. To cover more possible dimensions, we focus on even dimensions, increasing the systolic size N by a step size of 2.

2. **Frequency Modelling**: To accurately estimate the expected frequency of an $N \times N$ systolic array on the target device, we develop custom Tcl scripts to scan the entire SLR systematically. We synthesize and place a square array within boundaries defined by a square Pblock of DSPs for a given dimension and location on the SLR. This process is repeated for every valid location across the SLR regions on VU37P and VU9P in a staggering fashion. We then collect all the timing summary reports and extract the necessary timing information. This empirical data set allows us to predict the maximum F_{max} of any candidate systolic array on the VU37P and VU9P.

3. **Design Checkpoint Generation**: Our custom Tcl scripts synthesize, place and route all the systolic arrays under the out-of-context mode using the non-project flow in Vivado 2018.3. We generate a checkpoint file of the design after the routing stage. This file preserves the exact placement and routing paths of all the LUTs and hard blocks in the systolic array. We can then re-apply these files directly to assemble the final design containing multiple systolic arrays. This approach allows us to maintain the frequency performance of each partition from the frequency data collection phase to the final re-assembly stage. We can guarantee the feasibility of the final design since each of the partitions has been successfully placed and routed in a previous stage.

3.3.2 Stage 2 – Optimization

Approach to 2D Partitioning

At a high level, our approach to 2D partitioning is inspired by the 1D partitioning approach proposed in our previous work [12]. Given a partition count K, the evolutionary algorithm we use CMA-ES to solve the optimization problem by assigning layers' workload to partitions (layers allocation). In each iteration of CMA-ES, we perform greedy partition sizing (resource allocation) on each population. We iteratively assign more processing elements (PE) to the bottleneck partition until no spare PE is available. Using the pre-computed cycle and frequency models, we calculate the cost of each population in terms of the run-time of the bottleneck partition. We then feed the cost back into the CMA-ES algorithm, and it will generate a new generation of layer assignments based on the given score. In Figure 3.10, we present the high-level overview of both the layer and resource allocation, indicating the data flow between the two main loops in our optimizer.

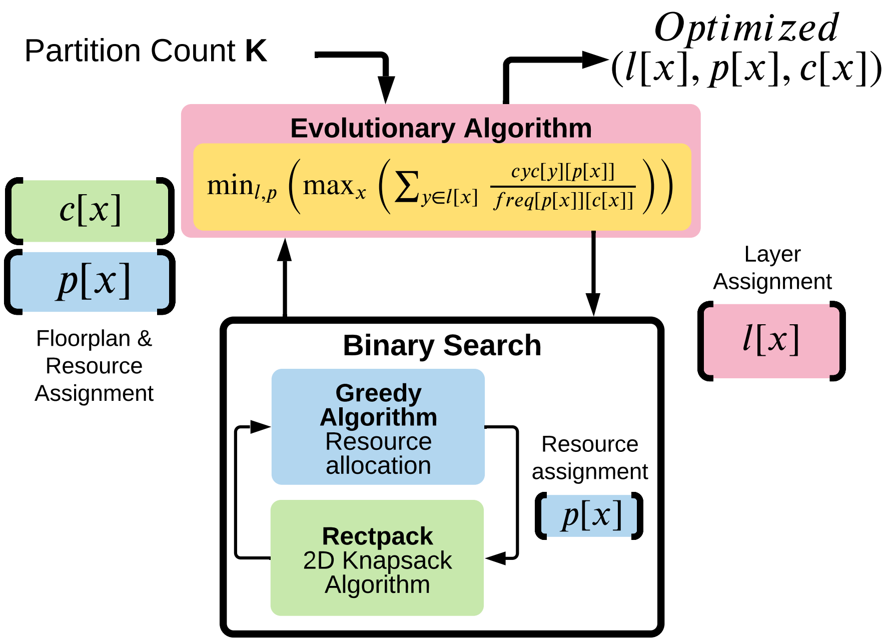

Partition Count K
Optimized $(l[x], p[x], c[x])$
Evolutionary Algorithm
$\min_{l,p}\left(\max_x\left(\sum_{y\in l[x]}\frac{cyc[y][p[x]]}{freq[p[x]][c[x]]}\right)\right)$
$c[x]$
$p[x]$
Floorplan & Resource Assignment
Layer Assignment
$l[x]$
Binary Search
Greedy Algorithm Resource allocation
Rectpack 2D Knapsack Algorithm
Resource assignment
$p[x]$

Figure 3.10: The optimization loop finds layer assignment $l[x]$, resource allocation $p[x]$, and placement allocation $c[x]$ for a partition size K.

The sizing of each partition depends on the two following components:

- **Binary Search**: We use a binary search to find the maximum number of PE units that yield a resource assignment that is packable using the maximal rectangles algorithm on the specified layout constraint. We exploit the monotonic relationship between the number of allocated processing units and the packability of the corresponding assignment to achieve an optimal result. We use the maximal rectangles algorithm due to its fast polynomial runtime and availability in an existing software package, `rectpack` [48]. For most packable solutions, some DSP wastage will be inevitable due to irregularities in the layout. The higher frequency of operation compensates for this. Furthermore, CMA-ES naturally decreases DSP wastage when solutions evolve to maximize throughput.

- **Square Partition Sizing**: We sweep across partition sizes in both dimensions with a step size of 2. This step size is selected since we allow double packing of two uint8 inputs on each DSP block, treating one DSP block as two systolic array PE units.

Variations of the 2D Partitioning Algorithm

We evaluate the following three variations of the 2D partitioning algorithm target design. The design shorthand name is shown in brackets.

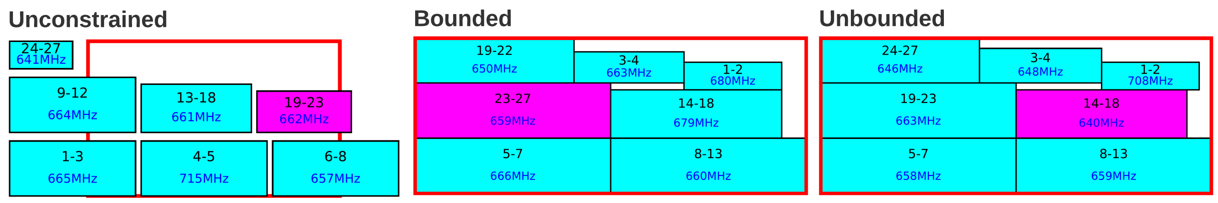
1. **Unconstrained Partition Sizing** (`Unconstrained`): To determine the upper bound for throughput and latency improvements achievable by square partitioning, we evaluate the case where the chip (i.e., the 2D knapsack bin) is infinitely sized for placement freedom. The only limitation is that the number of PE units must be less than or equal to 17280, the number of DSPs on a VU37P. For VU9P, the PE units must be less than or equal to 13680.

2. **DSP Placement Aware Sizing**: We use the maximal rectangles algorithm to determine whether a resource allocation is feasible and can be packed onto the chip.

   (a) **Unbounded Frequency** (`Unbounded`): When calculating the frequency of each systolic array, the optimizer only considers the array dimensions. The expected F_{max} for one systolic array dimension is calculated as the average frequency across all positions on the board. The best result generated by this variation of the optimization process does not have the most accurate performance representation when we realize these designs on the target FPGA chip. The solution generated by this variation is packable, but the throughput is usually lower than the `Unconstrained` variant.

   (b) **Bounded Frequency** (`Bounded`): When calculating the frequency of each systolic array, both the array's dimensions and the placement location, provided by the maximal rectangles algorithm, are considered. This variation's packable solution generates the most accurate performance metrics when the design is placed and routed on the FPGA.

Figure 3.11 compares packing plans generated by each of the variations for *MobileNets*. The red boundaries signify DSP grid boundaries, magenta squares represent the slowest bottleneck partition, and the cyan squares represent non-bottleneck partitions. All three PE grids presented here are subject to the same dimensions. In most cases, packing awareness reduces DSP utilization but produces routable implementations, while 100% DSP utilization usually leads to unroutable designs or designs operating at very low frequencies.



	Unconstrained	Unbonded	Bounded
Throughput (Img/s)	1795	1698 (-5.4%)	1800 (+0.27%)
DSP Wastage	5.11%	13.78%	15.04%

Figure 3.11: Comparison between solutions generated by `Unconstrained` (left), `Bounded` (middle), and `Unbounded` (right).

`Unconstrained` provides a non-packable design with the second-best throughput. However, these designs are usually not optimized and provide a poor performance estimation since the design is usually very congested and has no placement guidance. Therefore, solutions generated by this variation are not feasible and will fail the routing stage. In our experiment, the very few that pass the routing stage result in a design operating at a low frequency of ≈140 MHz. After enabling the maximal rectangles algorithm, we can see that all the partitions will nicely fit into the PE grid boundaries for design in `Unbounded` and `Bounded` designs. In this specific case, `Unbounded` can provide a packable design while suffering throughput performance loss, while `Bounded` design is packable and has the highest throughput out of the three. We observe that packing awareness reduces DSP usage but produces more valid and routable solutions by leaving resource space on other hard blocks and routing.

Chapter 4

Implementations

4.1 Systolic Array Implementation

This section provides a detailed description of the specific systolic array architecture implementation. We also introduce how we inferred the various hard blocks available from the Xilinx Ultrascale+ family to construct the components needed in the systolic array.

4.1.1 Systolic Array Architecture

We present an overview of the systolic array architecture used in this work in Figure 4.1 with the data flow. The green and blue arrows represent the data flow of the pixel matrix - A and the weight matrix - B, respectively. The red arrows represent the data flow of the output matrix - D. Each yellow block represents processing elements while the other blocks represent a mem block. The black arrows represent the data flow from the AXI interface to $s2mm$ and $mm2s$ to the AXI interface.

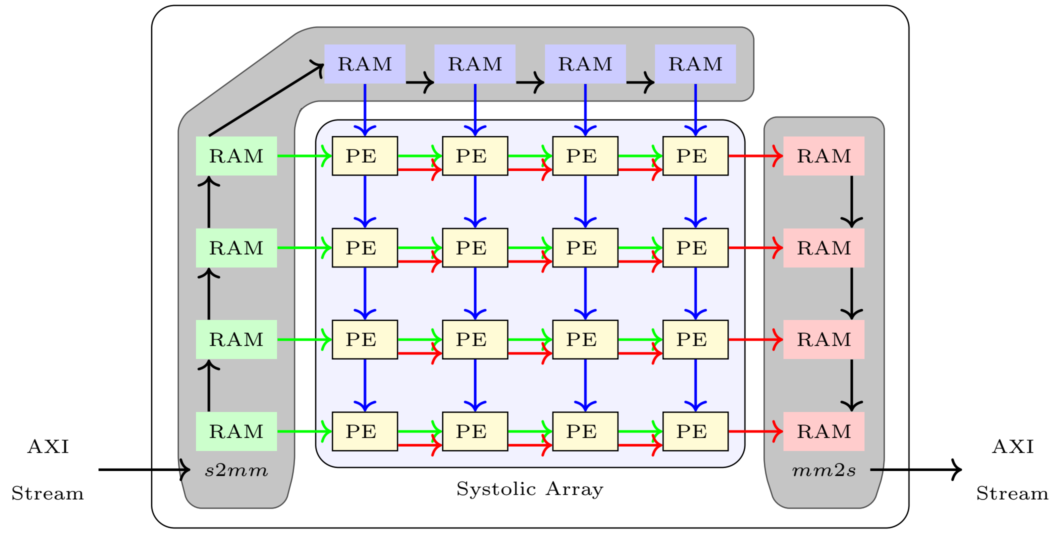


Figure 4.1: A high level overview of all the data flow in a 4×4 systolic array.

We code this implementation in a combination of SystemVerilog and Verilog. We develop and run testbenches in ModelSim 2020.1 to demonstrate the functional correctness of computing different matrix sizes and the first layer of GoogLeNetV1. We identify a matrix multiplication of $D = AB$ as having the pixel matrix A multiplied by the weight matrix B, giving us the result matrix D. There are four major components in our systolic array:

- **On-chip IO handling**: Each array has an *s2mm* component responsible for translating serial input into pixel and weight input responding to each row and column of the PE array matrix. In contrast, the *mm2s* component repacks output from each row back to one serial output. These components follow the reference AXI connection, allowing easy communication and utilizing RAMB18E2 available on Xilinx UltraScale architecture.
- **Computation handling**: Each array design can accommodate flexible square dimensions where each of the PE acts as multiply and accumulate (MAC) unit in a traditional systolic array design. We infer all the PEs as DSP48E2 hard blocks on VU37P and VU9P. Besides the simple input of one pixel per row, our computation array can also handle the double-packing of int8 pixel input.
- **Variable Pipeline between PEs**: To achieve high operating frequency, we have configurable pipeline stages between each PE block row-wise and column-wise. By

splitting the connection between each PE into multiple shorter paths, we can lower the routing difficulty significantly. This turns out to be one of the main contributors to retaining a high operating frequency even at larger dimensions.

- **Controller handling**: Each array has a controller unit to handle all the data coming to and from the AXI stream interfaces and the array. This configurable controller controls data movement from the IO unit to the computation unit by manipulating the reading and writing addresses to avoid excessive use of shift registers.

The final design generated from the optimizer will contain multiple partitions where each partition is a whole systolic array unit containing all the components mentioned above. Only two AXI interfaces from the first partition's $s2mm$ unit and the last partition's $mm2s$ connect to the die's I/O.

4.1.2 Systolic Array Coding Overview

We can split the major modules of the codebase into the *Verilog/System Verilog Code*, *Custom Scripts for database generation*, *Functional Simulation Scripts*, and *Vivado Specific Scripts*.

The *Verilog/System Verilog Code* module includes all the hardware codes used to construct the systolic array. The final tope level module `cpsa_mm_top.sv` combines multiple memory modules and the array module to achieve a fully functional matrix multiplication systolic array with I/O support.

The *Functional Simulation Scripts* module provides scripts allowing the user to easily configure the dimension of a systolic array and the target matrix to be computed for functional simulation. We can test the array and the array with memory support independently.

The *Custom Scripts for database generation* and *Vivado Specific Scripts* modules provide scripts allowing the user to automatically generate the frequency model by creating all the necessary files, including the placement constraint files and Verilog code to sweep over systolic arrays with different dimensions across the staggered location.

A more detailed overview of the codebase is attached in Appendix C.

CNN mapping to the hardware domain

Knowing that we can use matrix multiplication to replace the convolution operation, we design a systolic array that computes matrix multiplication effectively. We decide on an output-stationary square systolic array and handle the matrix multiplication with unsigned 8-bit Multiply-Accumulate (MAC) units. Each MAC unit will be abstracted as a *Processing Element* in the Verilog implementation. To realize the MAC unit, we decided to pack every two of them into one *DSP48E2* hardblock by splitting the DSP's I/O data port into upper and lower bits. To supply the array with pixel and weight data, we implement one *RAMB36E2* for every *DSP48E2* hardblock along the row and column. To retrieve the output of the systolic array, we add a column of *RAMB36E2* to store all the output from all the DSP hardblocks.

Processing Elements

We exploit the interface feature of SystemVerilog to create a public interface for inferring different processing elements. We abstract the usage of PE with a public interface in the `pe_interface` module. This allows developers to easily swap different internal implementations of the processing elements without affecting the overall architecture, which controls the input and output RAMs and the interconnection pipeline. Each PE in the systolic array is linked together using the general `PE_interface` containing the following parameters:

1. the type of input data to each processing element, DATA_TYPE,
2. whether the input data is signed, SIGNED,
3. the width of the input data, DATA_WIDTH,
4. the width of the output data, ACC_DATA_WIDTH,
5. the latency incurred in each processing element, LATENCY,
6. the number of input data from the channel A, NUM_A, and
7. the number of input data from the channel B, NUM_B.

For the current implementation and results presented in this work, we follow the settings in Table 4.1.

Table 4.1: Our parameter setting used

DATA_TYPE	SIGNED	DATA_WIDTH	ACC_DATA_WIDTH
UINT8	False	8	16
LATENCY	NUM_A	NUM_B	
4	2	1	

To infer the internal implementation, we also need to define the value of `PE_PLUGIN` through a header file. By defining the `PE_PLUGIN` as `xilinx_dsp48e2_double_uint8`, we perform a double packing of two UINT8 integers from the A channel, which multiply and accumulate with the single UINT8 from the B channel. Also, we use the `(* use_dsp48 = "yes" *)` keyword to infer each processing element as a DSP48E2 hard block on the FPGA die.

Block RAMs

To infer the internal implementation for each of the block memory, we use both the `(* use_bram = "yes" *)` and `(* ram_style = "block" *)` keyword to infer each `mem` module to a RAMB18 hard block. On top of the keyword, we also have to force the data width parameter to be 32. Failure to do so will result in configurable logic blocks being used instead of a dedicated RAMB18 hard block. This will negatively impact the max frequency achievable in the final place-and-route design in our experiments.

Controller

We embed a controller in the `cpsa_mm_top` module to accommodate different matrix dimensions. Firstly, the controller sends the correct read addresses at the correct cycle to each `mem` module in the `s2mm` module, considering the inter-row and inter-column pipeline between each PEs. Next, the controller needs to fire the init signal at the proper cycle, ensuring the accumulation in each processing element is correct. Lastly, the controller is responsible for resetting the array and getting ready for the following data patch.

4.2 Optimizer Implementation

This section provides a more detailed overview of the optimizer codebase consisting of three stages - 1) parser, 2) processor, and 3) output generation. Then, we focus on the processing stage to explain how we generate and update the layer and resource assignment. We also cover the necessary decoding and encoding detail on the format of layer assignment to interface with the CMA-ES interface. Finally, we cover how we generate the whole data set for further analysis.

4.2.1 Parser

On top of the cycle dataset - $cyc[][]$ and the frequency dataset - $freq[][]$, the optimizer need two more configuration files in YAML Ain't Markup Language (YAML) format.

The first configuration file contains a list of possible placement profiles. These profiles are essential for configuring the KnapSack solver in the processing stage. We created each profile by referencing the datasheet of the specific FPGA dies, considering the physical placement of the DSP block on the die. We provide examples and guidance to create profiles in Appendix A.

The second configuration file is a general configuration that contains six different sections of parameters controlling how the optimizer behaves. We provide a more detailed explanation of each section in Appendix B.

4.2.2 Processor

The central insight here is that we split the processing into three steps (1) layer assignment, (2) resource allocation, and (3) running the Maximal Rectangles algorithm. This allows the search complexity of layer assignment to be decoupled from resource allocation and incorporated with frequency related-metrics into the cost function of each sampling. CMA-ES handles the layer assignment process and the resource allocation step cost in polynomial time. Once we have a fixed layer assignment $l[x]$, we can determine resource allocation $p[x]$ in a greedy, optimal manner. This is possible only because (1) we observe that the scaling trends of cycle count for each layer in the $cycles[][]$ cycle dataset are monotonically decreasing as a function of systolic array size, and (2) we focus on minimizing the maximum cycle count across all partitions. After defining the layer assignment $l[x]$ and the resource allocation $p[x]$, the **RectPack** library handles the packable check. Finally, the location assignment $c[x]$ is used to search for the operating frequency of each partition. Depending on the variation of the optimizer, we calculate the final cost scores based on $l[x]$, $p[x]$, and $c[x]$ and return them to CMA-ES for continued optimization.

Layer Assignment translation

To translate the discrete layer assignment $l[x]$ into an N-dimension real-value vector suiting the sampling format of CMA-ES, we construct $l[x]$ as a list of sizes $k-1$ for a k grouping layer assignment. Each value in the list has to be a real value ranging from 0 to 1. Each value represents the consecutive percentage of layers being assigned in each group. We multiply the real value with the total number of layers in a network and round off the value to an integer for each grouping. That integer will then represent the number of layers in that grouping. To calculate the number of layers assigned to the last grouping, we assign all remaining layers to the last group. An example of such encoding conversion is provided in Equations 4.1 to 4.3. Since there are only $k-1$ real values, this allows us to reduce the complexity of the problem by one dimension.

$$\text{Assuming } k = 3 \text{ and Number Of Layers } = 58, \tag{4.1}$$

$$|l| = (k-1) = 2, \tag{4.2}$$

$$\text{if } \mathit{sampling} = [0.27, 0.27], \text{ then l[x]} = [15, 15, 28] \tag{4.3}$$

Inevitably, some layer assignment encodings cannot be translated properly. We define an illegal layer assignment encoding if either condition is met:

1. the sum of layer count does not equal the number of layers in the network and
2. the amount of layer assigned to any group equal to zero.

To guide CMA-ES to navigate to explore legal layer assignments, we add a penalty offset to the cost of each illegal layer assignment found. This mechanism allows CMA-ES to avoid illegal sampling in early iteration and focus more iterations on exploring legal solutions yielding better-optimized results.

Resource Allocation

After the CMA-ES sampling provides a layer assignment, we compute resource allocation in polynomial time. Once we know which set of layers are assigned to which partition $l[x]$, we determine resource allocation $p[x]$ in a greedy, optimal manner. We add one step of processing elements to the partition with the highest cycle count. The iterative process of resource allocation is illustrated in Algorithm 3.

Algorithm 3: Overview of the Greedy Sizing Algorithm

```
$PEStep = ProcessingElementStep\ p[x] = (PEStep, ..., PEStep)$;
$LimitFlag = False$;
while $!LimitFlag$ do
    $x' = \max_x(cyc[x])$ ;                    /* Find the bottleneck partition */
    $p[x'] += PEStep$ ;  /* Increase allocation to bottleneck partition */
    $cyc[x'] = \sum_{y \in l[x']} cycles[y][p[x']]$ ;   /* Recount cycles for partition $x'$ */
    if $\sum_{pe \in p[x']} pe^2 \geq MaximumProcessingElements$ then
        $LimitFlag = True$
    end
end
return $p[x]$;
```

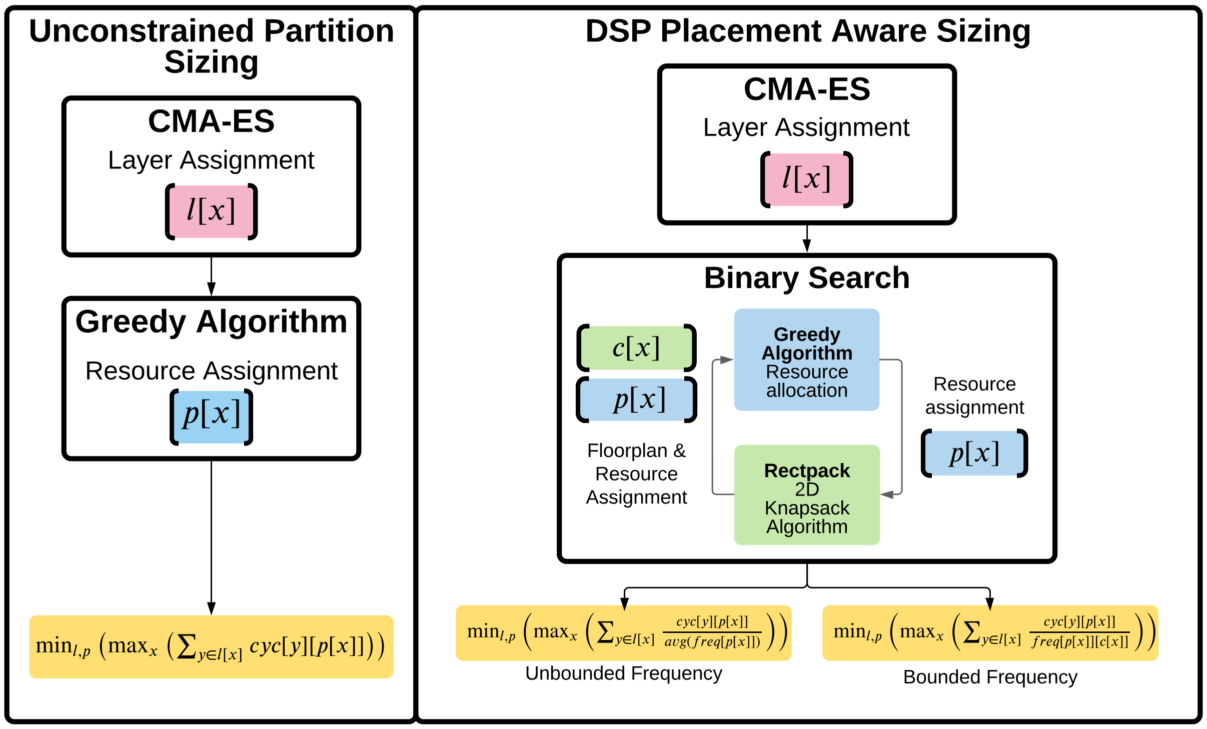
In the case of **DSP Placement Aware Sizing** optimization, we use binary search to look for the maximum amount of PE that is still packable. We reuse the greedy allocation in Algorithm 3 and check whether the resource allocation is packable using the *Maximal Rectangles Algorithm*. We then adjust the maximum number of PE in the allocation process until we reach a packable resource allocation and have the max amount of PE. The pseudocode of the binary search algorithm can be found in Algorithm 4.

Algorithm 4: Overview of the binary search algorithm

```
while $!converge$ do
    $lb \leftarrow 17280/2$ ;                /* Lower bound of the binary search */
    $up \leftarrow 17280$ ;                  /* Upper bound of the binary search */
    while $lb \neq up$ do
        $mid \leftarrow (lb + up)/2$;
        $p[x] \leftarrow GreedySizingAlgorithm(mid)$;
        $isPackable, c[x] \leftarrow RectPack(l[x], p[x])$;
        if $isPackable$ then
            $up \leftarrow mid$;
        else
            $lb \leftarrow mid$;
        end
    end
end
return $l[x], p[x], c[x]$;
```

Variation Flow

For variant **Unconstrained Partition Sizing**, the cost function only needs the layer assignment - $l[x]$ and resource allocation - $p[x]$ to calculate the cycle count in the bottleneck partition. We then feedback the cycle count to CMA-ES as the cost of each sampling. For variant **DSP Placement Aware Sizing with Unbounded Frequency**, the cost function requires the layer assignment - $l[x]$, resource allocation - $p[x]$. We determine the operating frequency of each partition by taking the average frequency of all arrays of the same size across different placement locations. For variant **DSP Placement Aware Sizing with Bounded Frequency**, the cost function requires the layer assignment - $l[x]$, resource allocation - $p[x]$, and placement location - $c[x]$. We determine the operating frequency of each partition by combining their placement location and systolic array size from the frequency dataset - $frequency[][]$ We then calculate the compute time of each partition as $cyclecount/operatingfrequency$ and feedback the compute time as the cost of each sampling for both **DSP Placement Aware Sizing** variations. We summarize all of these variations and present them in Figure 4.2



Unconstrained Partition Sizing
CMA-ES
Layer Assignment
$l[x]$
Greedy Algorithm
Resource Assignment
$p[x]$
$\min_{l,p}\left(\max_x\left(\sum_{y\in l[x]} cyc[y][p[x]]\right)\right)$
DSP Placement Aware Sizing
CMA-ES
Layer Assignment
$l[x]$
Binary Search
$c[x]$
$p[x]$
Floorplan & Resource Assignment
Greedy Algorithm Resource allocation
Rectpack 2D Knapsack Algorithm
Resource assignment
$p[x]$
$\min_{l,p}\left(\max_x\left(\sum_{y\in l[x]} \frac{cyc[y][p[x]]}{avg(freq[p[x]])}\right)\right)$
Unbounded Frequency
$\min_{l,p}\left(\max_x\left(\sum_{y\in l[x]} \frac{cyc[y][p[x]]}{freq[p[x]][c[x]]}\right)\right)$
Bounded Frequency

Figure 4.2: A progress representation on how $l[x]$, $p[x]$ and $c[x]$ are generated for each variation

4.2.3 Output Generation

To avoid writing conflicts on the resulting CSV file, we save statistics of each iteration in the runtime memory. Each run's related statistic is dumped on the dedicated CSV file according to the configuration file's 'PATH' section when the CMA-ES reaches the last iteration. In each iteration, we collect the following data:

1. Trial count,
2. Iteration (generation) count,
3. Name of the network,
4. Partition count (k),

5. Time spent in this iteration (seconds),
6. Layer assignment of the best design in this iteration ($l[x]$),
7. Resource allocation of the best design in this iteration ($p[x]$),
8. Placement location of each partition of the best design ($c[x]$),
9. Frequencies of each partition of the best design in this iteration,
10. Latencies of the best design in this iteration,
11. Compute times of the best design in this iteration in ms,
12. Throughput of the best design in this iteration in ms,
13. DSP utilization of best design,
14. LUT utilization of best design,
15. Throughput Gain compared to a fully mapped design, and
16. Latency Penalty compared to a fully mapped design.

To pick a specific design depending on their metrics, we develop analysis scripts using a python notebook to select 1) the most balanced design, 2) the design with the highest throughput, and 3) the design with the lowest latency. The notebook allows us to plot figures quickly and promptly dive into more in-depth analysis. Using a python notebook also demonstrates how we collect all the statistics and guide how end-users navigate all the results.

After picking a specific design, we combined all the DCP files for each partition. We used Vivado to generate the final design that contains all linked partitions as one complete design.

Chapter 5

Results and Discussion

In this chapter, we first analyze the overall performance of our optimizer, including its runtime and quality of result (QoR). Then, we analyze the routability and throughput improvement of the FPGA design implementation. Finally, we compare our results to state-of-the-art related work.

5.1 Optimizer Performance

5.1.1 Optimizer Runtime

We collect the runtime information on a machine equipped with Ryzen 9 5950x with 16 cores and 32 GB of DDR4 RAM. We have 3075 different flavors for both VU9P and VU37P. The dataset of VU9P takes 153 mins (∼2.5 hours), and the dataset of VU37P takes 210 mins (∼3.5 hours). These datasets include all the different Knapsack settings and neural network targets mentioned in Section 2.1. The optimizer needs to process 100 samplings in each iteration, including the layer allocation translation, greedy sizing algorithm, and the maximal rectangular algorithm depending on the variations.

We summarize the runtime data of each iteration in our optimization flow covering all the 6150 different flavors in Figure 5.1. We group the runtime data by their respective KnapSack setting and organize the data spread using a Box Plot in Figure 5.1. We can see many outliers in our data set as represented. These outliers can add up to 200 seconds of runtime per iteration, and we filter these outliers in the follow-up analysis in Figures 5.2 and 5.3. One of the biggest reasons the dataset contains many outliers is that when the value of k is high enough; it takes more iterations in the binary search process to reach a packable design. The optimizer then needs to run the maximal rectangular algorithm multiple times to decide the placement of those k partitions.

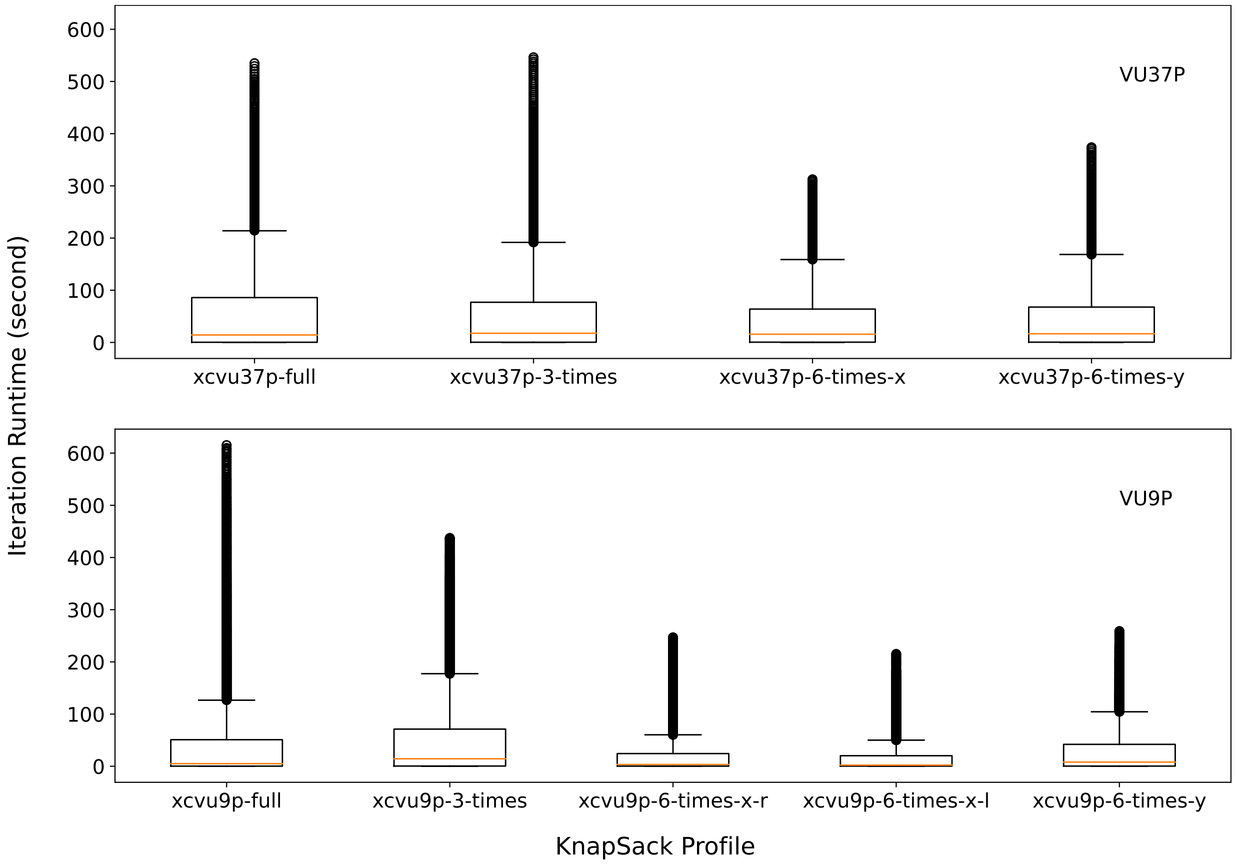


Figure 5.1: Iteration runtime against different KnapSack profiles

In Figures 5.2 and 5.3, we clean up the dataset and present the average runtime of each iteration against different KnapSack profiles and neural networks. In Figure 5.2, the iteration runtime trend matches the number of available DSPs in each of the profile, where the largest knapsack profile `xcvu37p-full` and `xcvu9p-full` require 60 seconds on average per iteration.

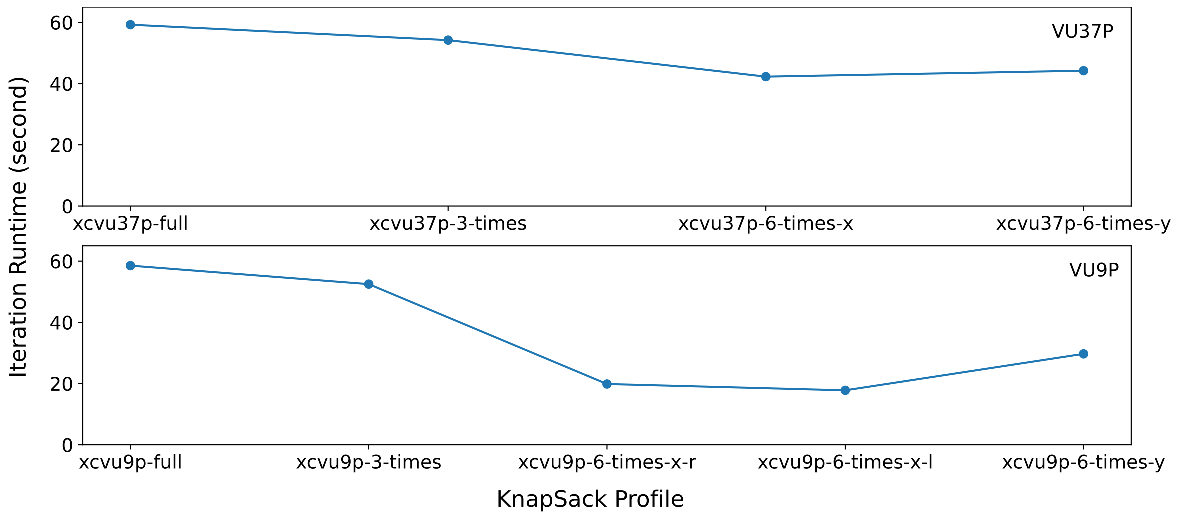


Figure 5.2: Average Iteration runtime against different KnapSack profiles

In Figure 5.3, the iteration runtime trend also matches the number of layers in a neural network. When the network is shallower, the permutation of possible layer groupings and the maximum k value is small.

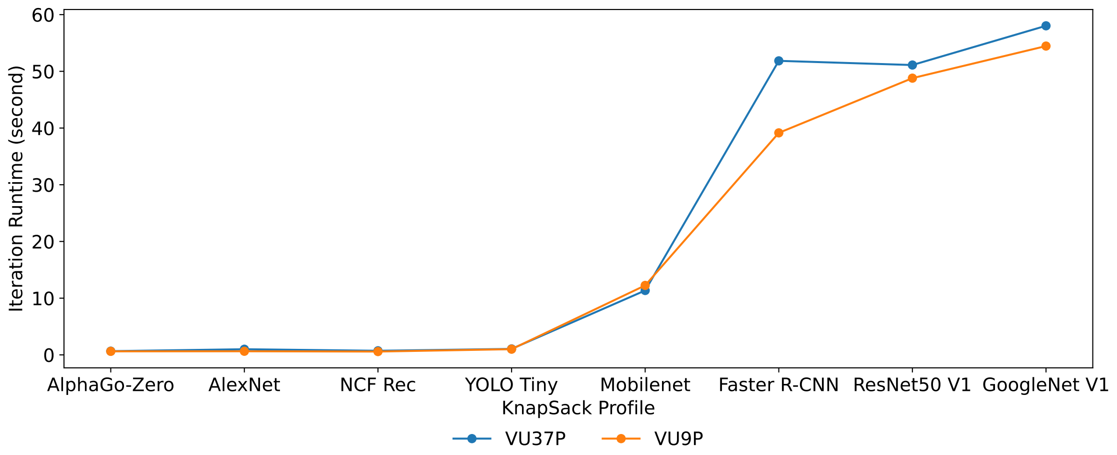


Figure 5.3: Average Iteration runtime against different networks

We can see that both the dimension of the 2D KnapSack profile and the number of layers in the target network together increase iteration runtime since both contribute to the complexity of the optimization problem space.

Another interesting finding is that we can train the CMA-ES algorithm to focus on legal solutions by adopting the penalty offset mechanism to enforce the legal encoding mentioned in Section 4.2.2. In Figure 5.4, we present the percentage of legal samplings per iteration. The data presented in the plot are from the optimization run on *Faster-RCNN* with `xcvu37p-6-times-y` and `xcvu9p-6-times-y` where $10 \leq k < 22$. We can see that even most runs start with no legal sampling. Once the population discovers a legal sampling, the algorithm can rapidly increase the percentage of legal sampling in the population. We can confirm that our penalty offset mechanism allows the optimization to spend more iterations with a population where 80% of its samplings are legal. In lower k values, the optimizer can achieve nearly 100% legal sampling in its later iterations.

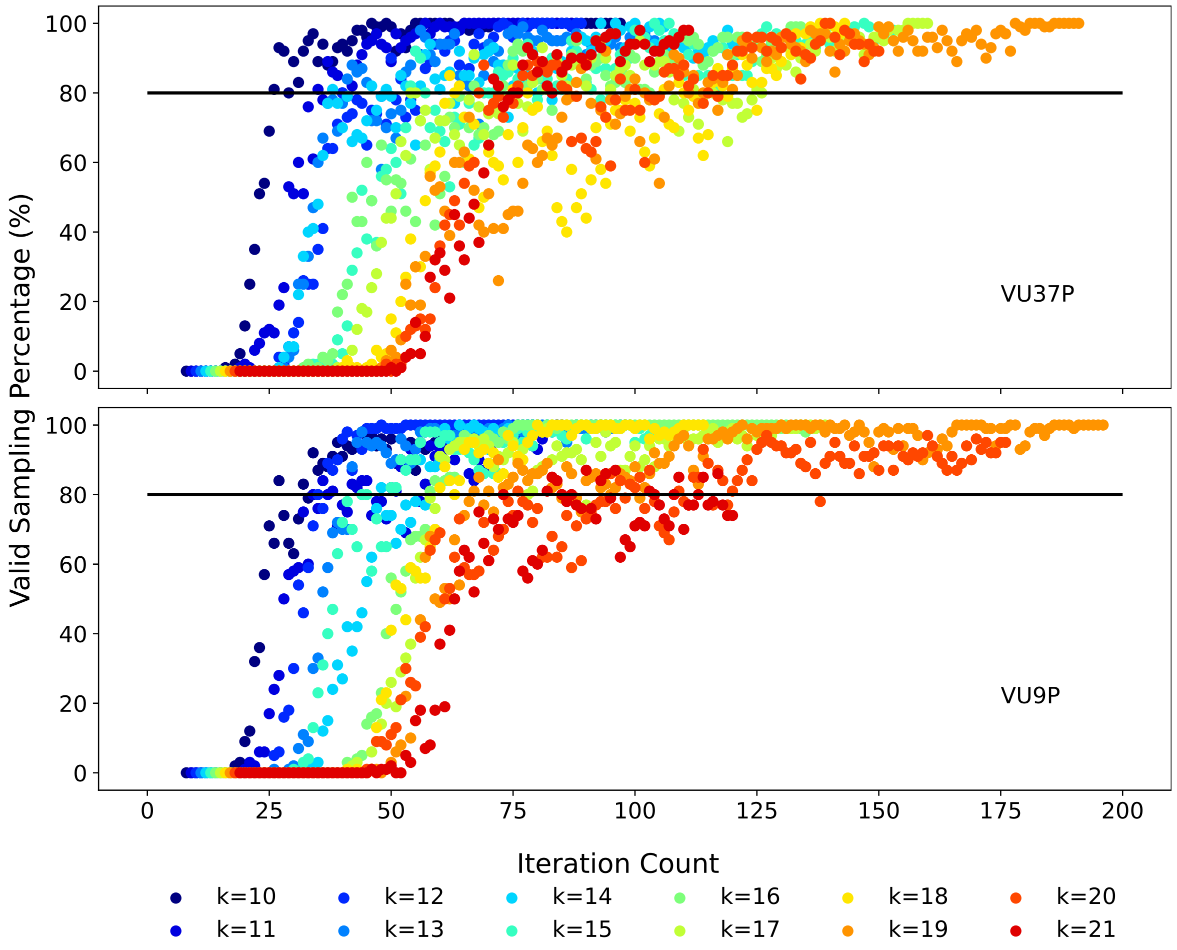


Figure 5.4: Legal design percentage per iteration trend

5.1.2 Design Quality

To evaluate the quality of our design, we define a baseline designs as the design that uses all the DSPs available in a given grid for one systolic array as the `fully-mapped design`. The `fully-mapped design` operates in a non-pipelined fashion, processing each layer one by one. We then calculate the throughput gain and latency penalty of every design against the throughput and latency of the `fully-mapped design`.

Overall Solution Quality Metrics

Table 5.1 and Table 5.2 shows the performance of the best non-replicated and replicated full-chip solutions generated by our partitioning algorithm on VU37P and VU9P, respectively. Designs in Table 5.1 and Table 5.2 were selected due to having the highest throughput, lowest latency, or most significant throughput gain to latency penalty ratio. These tables illustrate the upper bounds of performance realizable with our workflow and achievable performance gains using smaller replicated designs. We observe that these smaller-scale replicated designs achieve up to 4× higher throughput, albeit at a higher latency than those using all 3 SLRs.

Table 5.1: Solution Performance on VU37P

		Non-Replicated			6x-Replicated		
Topology	Selection	K	Tput (img/s)	Lat. (ms)	K	Tput (img/s)	Lat. (ms)
`AlexNet`	Max Tput.	8	280	28.48	8	1098	43.7
	Min Lat.	3	261	11.46	3	1066	16.88
	Balanced	3	261	11.46	3	1066	16.88
`AlphaGoZero`	Max Tput.	8	5820	1.37	8	16481	2.91
	Min Lat.	3	5031	0.6	3	15014	1.2
	Balanced	4	5820	0.69	3	15014	1.2
`Faster R-CNN`	Max Tput.	17	1206	14.09	19	2284	49.89
	Min Lat.	3	446	6.71	3	1609	11.19
	Balanced	8	785	10.18	3	1609	11.19
`GoogLeNetV1`	Max Tput.	29	2993	9.69	18	6035	17.89
	Min Lat.	3	1001	2.99	3	3903	4.61
	Balanced	9	1924	4.68	3	3903	4.61
`MobileNet`	Max Tput.	22	3582	6.14	22	10488	12.58
	Min Lat.	3	1003	2.99	3	4752	3.79
	Balanced	17	3347	5.08	3	4752	3.79
`NCF Rec.`	Max Tput.	8	1385	5.78	8	9092	5.28
	Min Lat.	4	1383	2.89	4	9085	2.64
	Balanced	4	1383	2.89	4	9085	2.64
`ResNet-50 V1`	Max Tput.	26	1436	18.1	26	2259	69.06
	Min Lat.	3	385	7.79	3	1384	13.0
	Balanced	9	712	12.63	3	1384	13.0
`Tiny-YOLO`	Max Tput.	9	879	10.23	7	2911	14.43
	Min Lat.	4	859	4.65	3	2373	7.58
	Min Lat.	4	859	4.65	3	2373	7.58

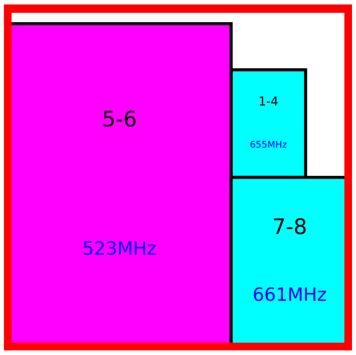
Table 5.2: Solution Performance on VU9P

		Non-Replicated			6x-Replicated		
Topology	Selection	K	Tput (img/s)	Lat. (ms)	K	Tput (img/s)	Lat. (ms)
`AlexNet`	Max Tput.	12	324	37.02	20	1639	73.2
	Min Lat.	6	298	20.12	3	1149	15.66
	Balanced	6	294	20.37	3	1149	15.66
`AlphaGoZero`	Max Tput.	8	1404	5.7	3	12221	1.47
	Min Lat.	3	1268	2.36	3	12221	1.47
	Balanced	3	1268	2.36	3	12221	1.47
`Faster R-CNN`	Max Tput.	12	324	37.02	20	1639	73.2
	Min Lat.	6	298	20.12	3	1149	15.66
	Balanced	6	294	20.37	3	1149	15.66
`GoogLeNetV1`	Max Tput.	8	836	9.57	29	4216	41.27
	Min Lat.	3	355	8.44	3	2845	6.33
	Balanced	8	836	9.57	3	2845	6.33
`MobileNet`	Max Tput.	25	3358	7.44	24	8408	17.13
	Min Lat.	14	2550	5.49	3	3831	4.7
	Balanced	12	1489	8.06	6	5249	6.86
`NCF Rec.`	Max Tput.	8	734	10.88	8	4342	11.05
	Min Lat.	4	734	5.44	3	4333	4.15
	Balanced	4	734	5.44	3	4333	4.15
`ResNet-50 V1`	Max Tput.	27	1008	26.77	24	1708	84.3
	Min Lat.	11	760	14.46	3	1006	17.88
	Balanced	9	359	25.06	3	1006	17.88
`Tiny-YOLO`	Max Tput.	6	270	22.22	8	2163	22.18
	Min Lat.	3	226	13.23	3	1650	10.91
	Balanced	3	226	13.23	3	1574	11.43

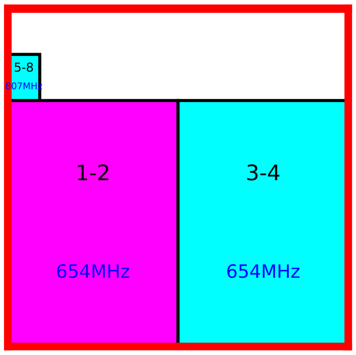
Solution Floorplans Overview

Figure 5.5 shows the designs generated by our partitioning algorithm for each topology on VU37P under the `xcvu37p-6-times-y` profile. Figure 5.6 shows the designs generated by our partitioning algorithm for each topology on VU9P under the `xcvu9p-6-times-y` profile. Each cyan block represents a partition, and the magenta block represents the bottleneck partition. A *bottleneck partition* is a partition that takes the longest execution time to finish the workload assigned to it. Since data can only move into the next partition when it finishes up its workload, the bottleneck partition ultimately determines the final throughput performance of the whole pipeline. The white space represents unused DSPs. The number on each block is the set of layers mapped to that partition. For example, in Figure 5.5(a), the first four layers of AlexNet are assigned to the cyan partition with 1–4 on it; the fifth and sixth layers are assigned to the magenta bottleneck partition with 5–6 on it, and the remaining layers are assigned to the cyan partition with 7–8 on it.

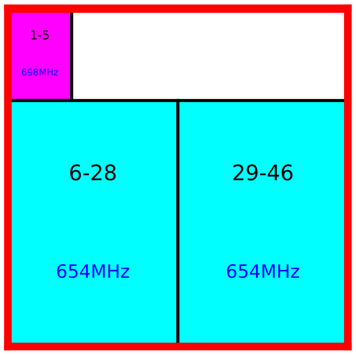
In Figures 5.5 and 5.6, the optimizer discovers that only 3–7 partitions achieve the most balanced performance. Many floorplans waste ≈18–29% of available DSPs to trade for higher operating frequency. We observe that the full-chip designs of `MobileNet` and `ResNet-50 V1` on VU37P result in smaller wastage on available DSPs at 17% and 10%, respectively, as shown in Figure 5.5. We hypothesize that this is due to the relatively more balanced workload across the network, allowing our optimizer to generate better designs at a higher partition count. This allows more room for better placement optimization leading to higher DSPs utilization in the same grid.



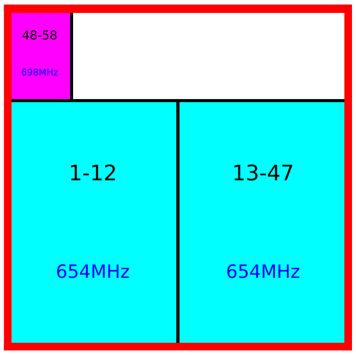
(a) AlexNet



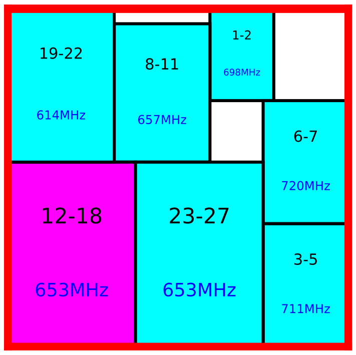
(b) AlphaGoZero



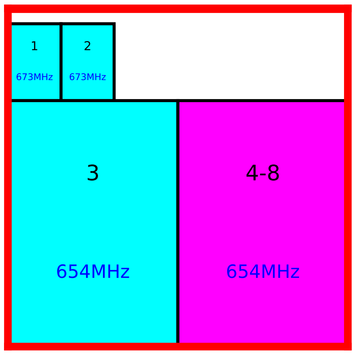
(c) FasterRCNN



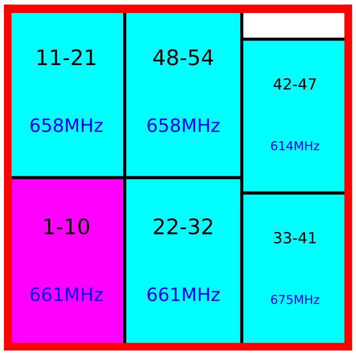
(d) GoogLeNetV1



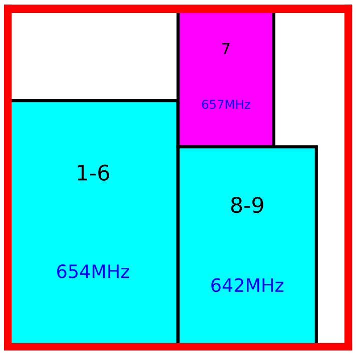
(e) MobileNet



(f) NCF Rec

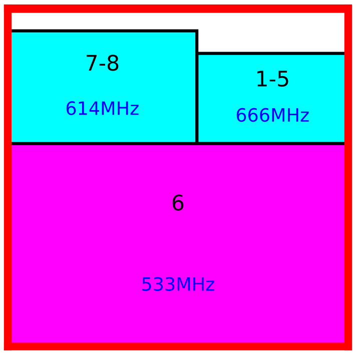


(g) ResNet-50 V1

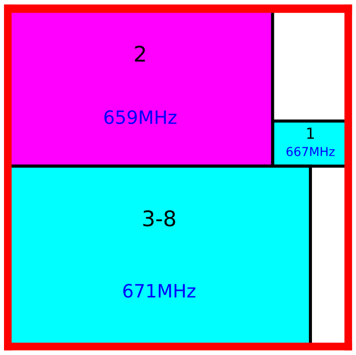


(h) YOLO Tiny

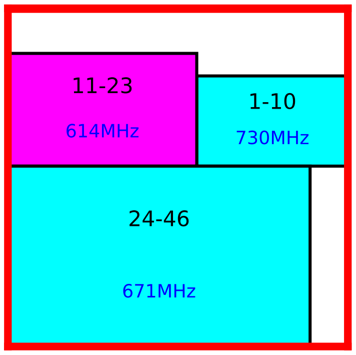
Figure 5.5: Partitioned solutions of every topology with the `xcvu37p-6-times-y` profile on VU37P



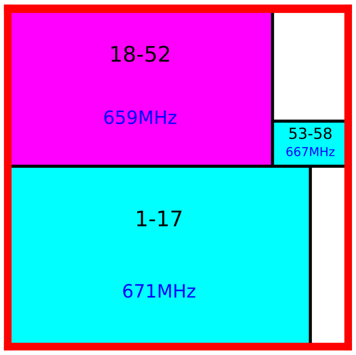
(a) AlexNet



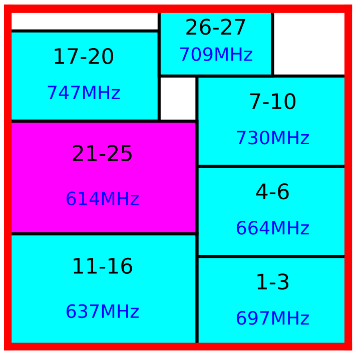
(b) AlphaGoZero



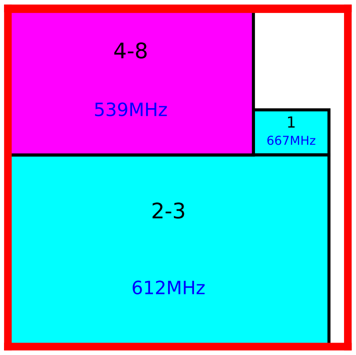
(c) FasterRCNN



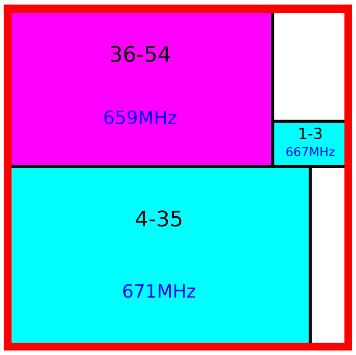
(d) GoogLeNetV1



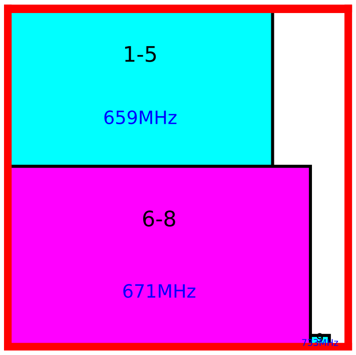
(e) MobileNet



(f) NCF Rec



(g) ResNet-50 V1



(h) YOLO Tiny

Figure 5.6: Partitioned solutions of every topology with the `xcvu9p-6-times-y` profile on VU9P

Balancing Throughput Gain and Latency Penalty

Figures 5.7 and 5.8 explore the relationship between throughput and latency for each topology targeting VU37P and VU9P as we increase the partition number in each design. Each data point represents an increment of one partition. We compute the throughput gain and latency penalty ratios, which compare our partitioned solutions' throughput (in img/s) and latency (in seconds) to the `fully-mapped design`.

Firstly, none of the generated solutions have a lower latency or throughput than the fully-mapped systolic array.

Secondly, we observe a monotonic increase in throughput for all network topologies and partitioning algorithm variations as the number of partitions increases until a certain point. There exists a partition count above which latency increases with negligible improvements in throughput for all topologies. We can find this optimal partition count in Figure 5.7 by tracing each plot line up to the point where the slope of the line is approaching infinity, or in other words, a vertical line. Forcing the design to have more partitions after this optimal point results in a negative trade-off where the latency penalty outweighs our throughput gains.

Thirdly, we confirm that deeper networks can achieve better throughput gains in higher partition count than shallower networks. We can achieve relatively high throughput gains for deep networks like `MobileNet` and `GoogLeNetV1` (more than 20×). Due to insufficient parallelizability, shallow networks like `AlexNet`, `AlphaGoZero`, and `Yolo-Tiny` cannot achieve such high throughput gains. The shallower networks reach saturation at low partition counts (3–4), while deeper networks reach saturation at much higher partition counts (13+). This demonstrates that deeper networks benefit more from our partitioning. Similar trends exist for all three algorithm variations.

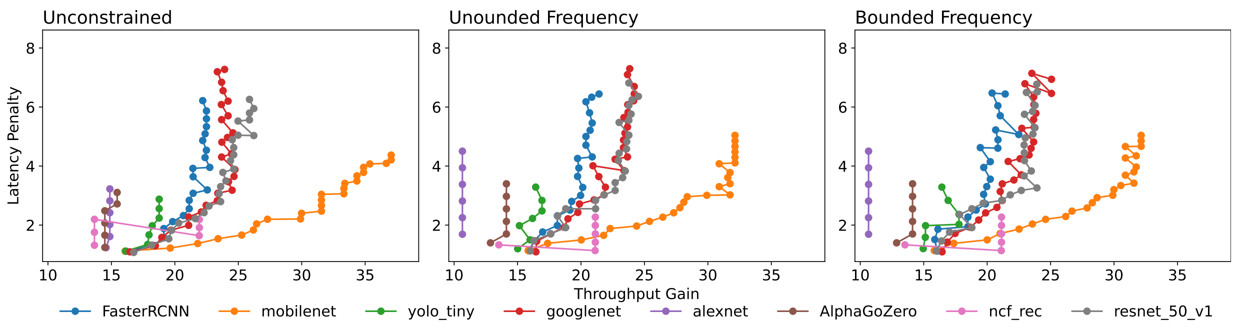


Figure 5.7: Latency penalty vs. Throughput gain by partitioning algorithm variation on VU37P.

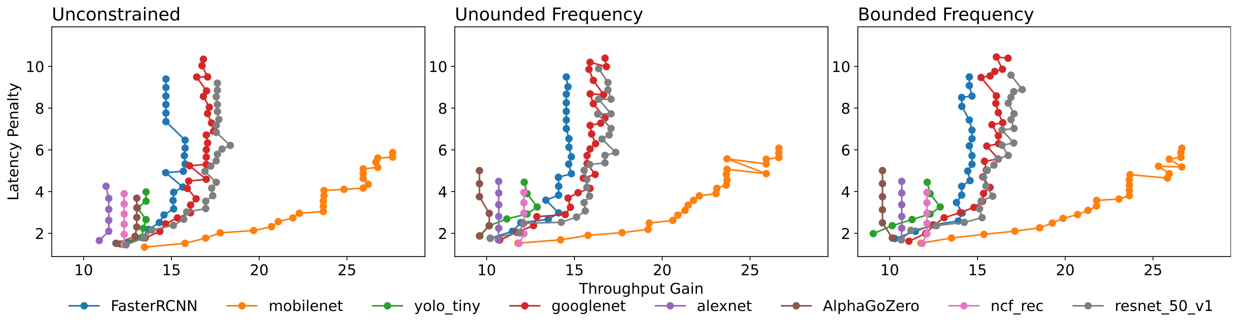


Figure 5.8: Latency penalty vs. Throughput gain by partitioning algorithm variation on VU9P.

To quantify performance that balances throughput and latency, we calculate the ratio between throughput gain and latency penalty for each solution. Figures 5.9 and 5.10 show the best solution generated by each algorithm variation for each topology based on this ratio. In other words, we pick the design achieving the most balanced design in Figures 5.7 and 5.8 and present them in a bar chart format.

For most topologies, the unconstrained variation of the algorithm generates better solutions than the **DSP Placement Aware Sizing** algorithms. This difference in performance shows that resource wastage caused by the packing process decreases throughput and increases latency. Counter-intuitively, but favorable to our proposed approach, for `ResNet-50 V1`, we find the best solution in terms of throughput gain to latency penalty ratio in the bounded variation of the partitioning algorithm. We hypothesize that this is due to the solution's relatively high number of partitions, as seen in Figure 5.5. This allows the algorithm more freedom to choose where to place the partitions; it can achieve superior placements concerning frequency.

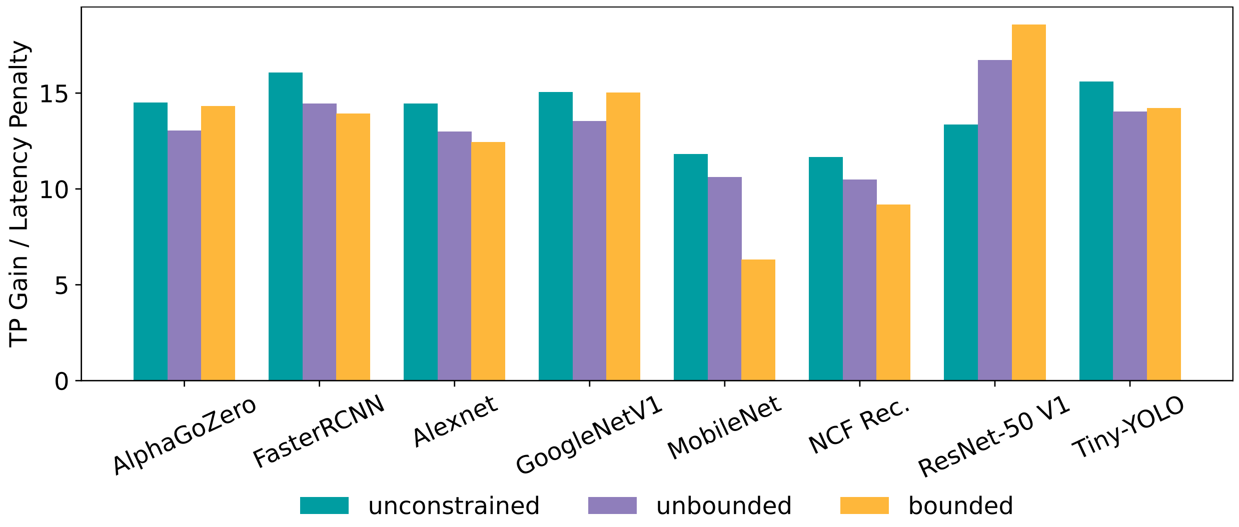


Figure 5.9: Ratio of Throughput gain to Latency penalty as a partitioning algorithm variation on VU37P.

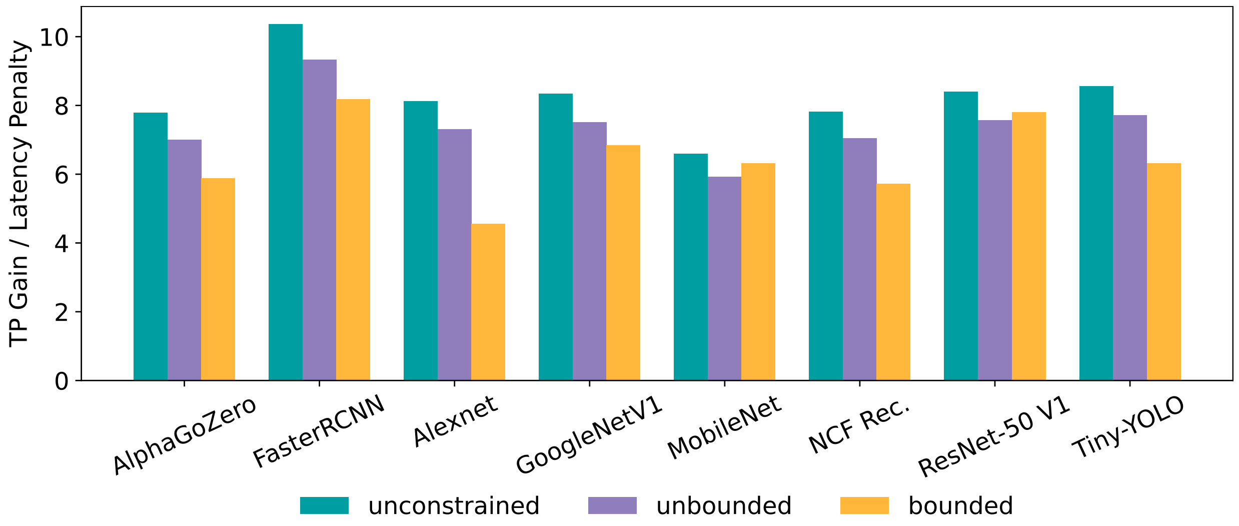


Figure 5.10: Ratio of Throughput gain to Latency penalty as a partitioning algorithm variation on VU9P.

5.2 FPGA Implementation Result

5.2.1 Routability

Figure 5.11 shows a comparison of one of our software-generated partitioning workflow floorplans and its placed and routed implementation on the VU37P. We restrict the available area of this design to half of the SLR on VU37P horizontally. The performance in Figure 5.11 is placed in the left half of the SLR0 region. The awareness of DSP locations in the partitioning algorithm allows us to successfully map processing elements to only half of the SLR0 region on the VU37P. With double inter-processing element pipeline stages, the complete `GoogLeNetV1` design uses 90% of DSP48E2s, 59% of CLB LUTs, and 51% of CLB registers.

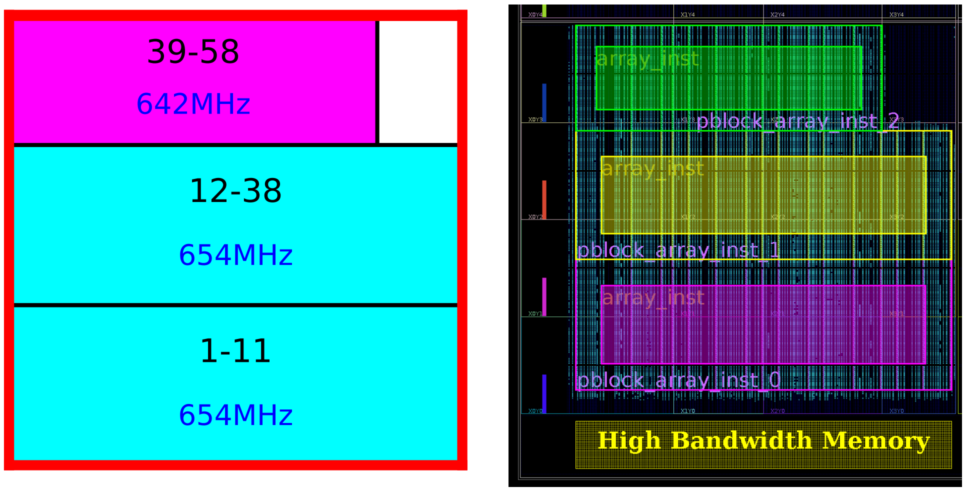


Figure 5.11: A side-by-side comparison of software floorplan (left) and Vivado's placed and routed implementation on VU37P (right) with `GoogLeNetV1`.

Figure 5.12 shows a comparison of one of our software-generated floorplans with its placed and routed implementation on the VU9P. We restrict the available area of this design to half of the SLR on VU9P vertically. The implementation in Figure 5.12 is placed in the lower half of the SLR0 region. The full `GoogLeNetV1` design uses 82% of DSP48E2s, 45% of CLB LUTs, and 42% of CLB registers.

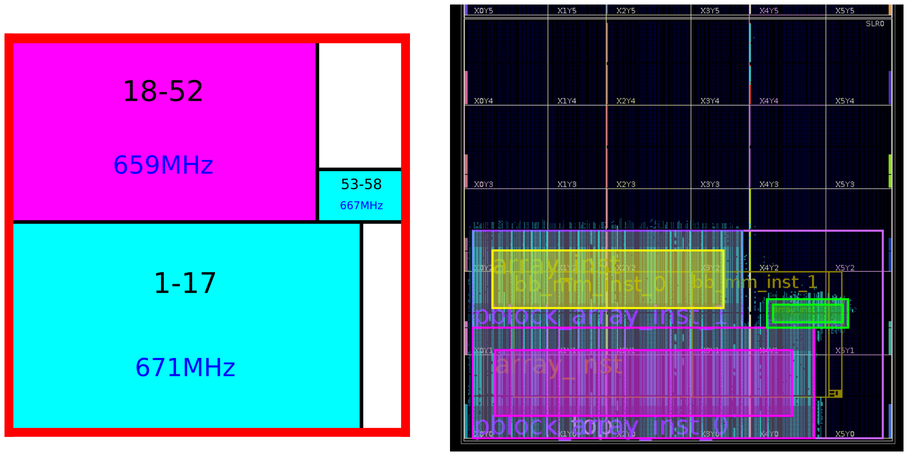


Figure 5.12: A side-by-side comparison of software floorplan (left) and Vivado's placed and routed implementation on VU9P (right) with `GoogLeNetV1`.

One of the significant obstacles to implementing these designs is routing congestion. While the 2D knapsack accounts for DSP placement, the physical implementation must consider register usage, the placement of LUTs, CLBs, and the local memory blocks. Figure 5.13 shows the trade-off between operating frequency and systolic array dimension. We observe that frequency degradation as a larger systolic array passes place and route. Vivado needs to go through several Global Rip-up and Reroute stages, even for the smaller systolic arrays. Changes like the number of inter-processing pipeline stages have a large impact on the overall F_{max} of the design. This trend is observed as the partition size increases. While utilizing 100% DSPs of an SLR region may intuitively seem like the optimal approach for maximizing system throughput, our observations show that a single systolic array approaching 64×90 pass place and route exhibits significantly lower F_{max} at ≈ 140 MHz. These observations reaffirm that smaller, replicated systolic array designs have much higher operating frequency, delivering better throughput.

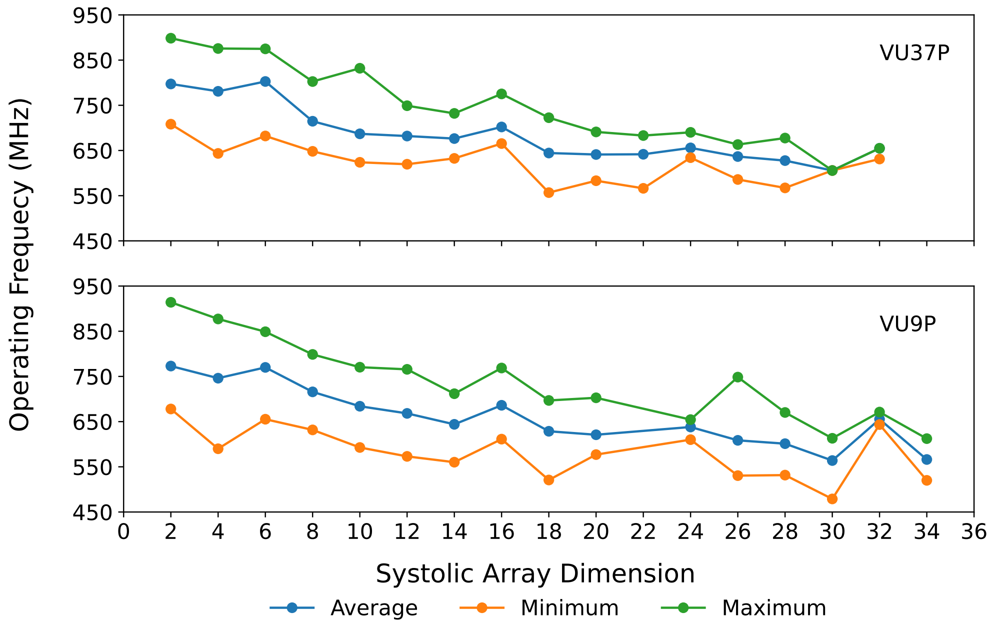


Figure 5.13: Systolic Array Dimension vs. Operating Frequency on VU9P (bottom) and VU37P (top). An overall 20.6% and 26.7% frequency drop can be observed in VU37P and VU9P, respectively.

Furthermore, our framework generates systolic array designs that operate at a high frequency than other tools such as AutoDNNchip [63] at 220 MHz and AutoSA [54] at 300 MHz. Our optimization generates designs favoring systolic array operating in the range of 550–670 MHz.

5.2.2 Demo on PYNQ

PYNQ is an open-source project from Xilinx, allowing developers to create a high-performance application on the Zynq, Zynq UltraScale+, and the Alveo platform. It combines a lot of IPs and interface into Python libraries and opens the possibility for hardware developers to present their designs to be used by the broadest possible audience.

In this demo, we use Jupyter Notebook as the software interface to interact with the systolic array to accelerate the operation in the first convolution layer of GoogleNetV1 via our proposed architecture in the previous sections. We construct the demonstration using Xilinx's **Py**thon productivity for Zy**nq** (PYNQ) Z2 development board from the

TUL Corporation, as shown in Figure 5.14. This board also come with a 650MHz dual-core Cortex-A9 processor handling all the network connection and hosting the Jupyter Notebook web server under a Linux kernel as a System-On-Chip (SOC) system.

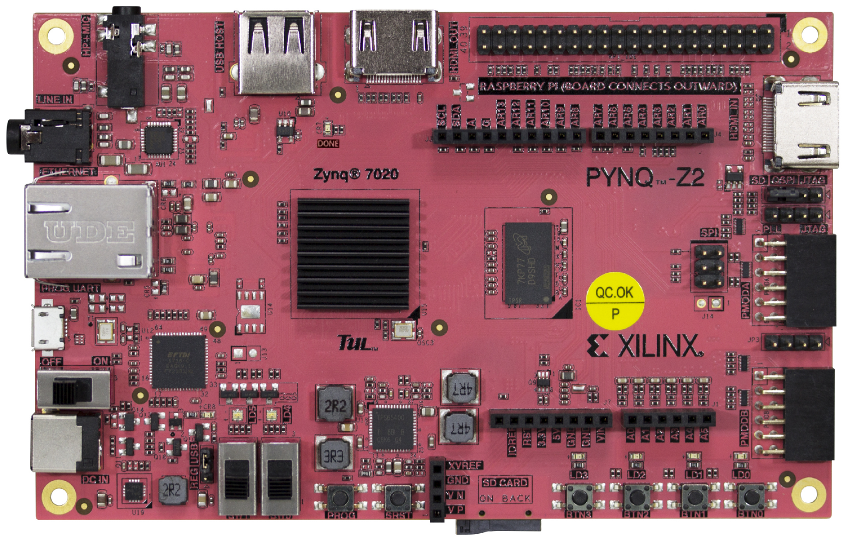


Figure 5.14: An PYNQ Z2 development board from TUL Corp. [13]

The significant difference of the FPGA chip on the PYNQ-Z2 board is that the DSP hard blocks are *DSP48E1* from the Xilinx 7-series platform. The 220 *DSP48E1*s have a smaller multiplier where it can only perform a 25×18 multiplication. This restricts our systolic array dimension up to 20×20. To verify and test our design on the PYNQ-Z2 board, we need to synthesize an *overlay* with the systolic array and the AXI controller, as illustrated in Figure 5.15. In the PYNQ terminology, an overlay is the programmable FPGA designs developers can create to accelerate the software application. We then load the overlay in the Jupyter Notebook to interact with the systolic array in a python environment. We present a high-level system overview of the whole demo in Figure 5.16.

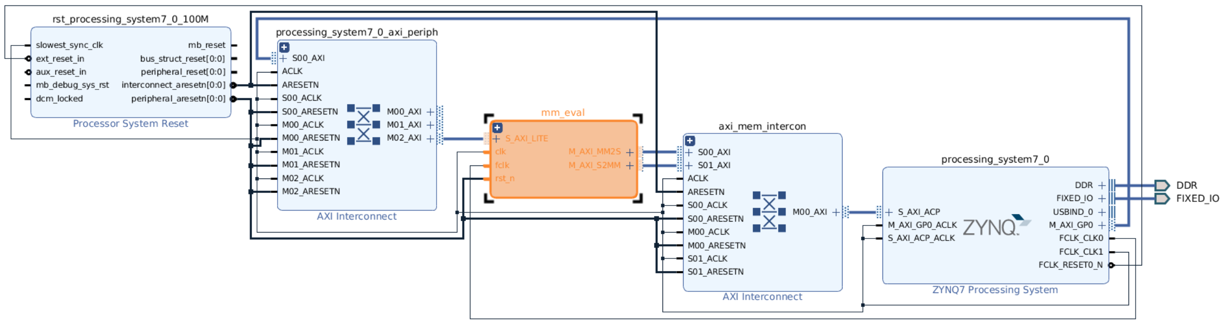


Figure 5.15: A overview of the customized systolic array and AXI controller overlay

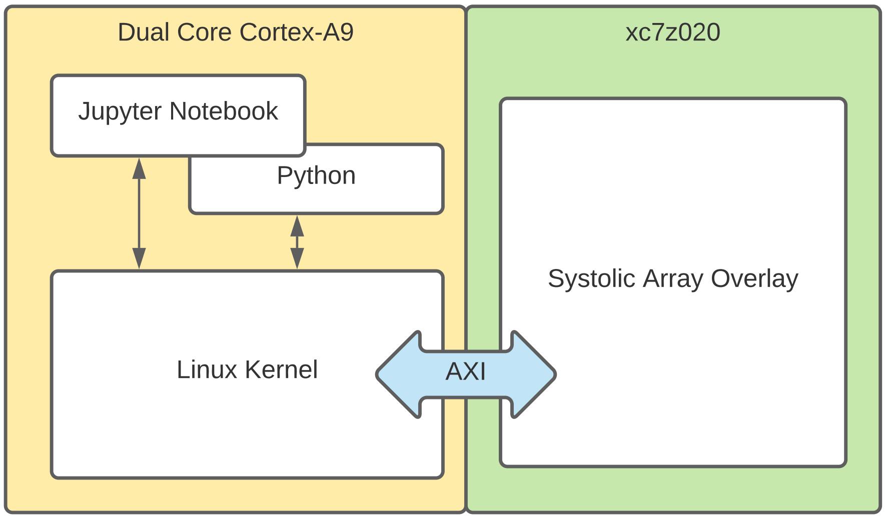


Figure 5.16: A high-level system overview of the PYNQ-Z2 demo

We synthesize our systolic array design with different dimensions and accelerate the operations on the first convolution layer of GoogleNetV1. We record the exact cycle count needed to finish the operation on each dimension and compare our data against the estimated cycle count obtained from SCALESim. We summarize the comparison in Figure 5.17.

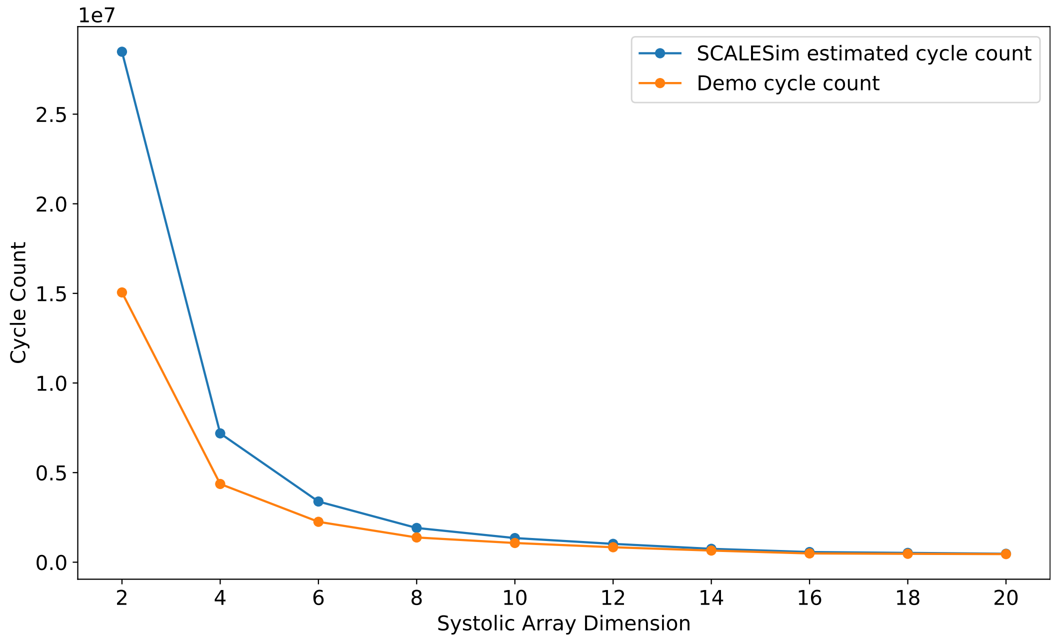


Figure 5.17: SCALESim estimation and actual systolic array operating cycle count against systolic array dimension

In Figure 5.17, we can see that SCALESim estimation overshoots when the dimension is relatively small. However, the estimation becomes more accurate to the cycle count record in the actual implementation after the array dimension is higher than a 10×10 array. We can also see that the actual cycle count is always smaller than the estimation from SCALESim.

Chapter 6

Conclusions & Future Directions

6.1 Conclusions

We present an algorithm partitioning Xilinx FPGA capacity across multiple square systolic arrays and boosting neural network inference throughput by up to 32.5× compared to a single unpartitioned systolic array. This algorithm uses CMA-ES to assign layers to systolic array partitions and a novel resource allocation algorithm that uses a 2D knapsack algorithm to size these partitions such that each partition achieves an optimal placement. We also present a complete workflow that generates performance-optimized, frequency-aware floorplans that can be routed on the Xilinx VU37P and VU9P FPGA. By collecting frequency data for systolic arrays at various locations on the die, our workflow can optimize for compute time and achieve operating frequencies of between 550 and 670 MHz. When compared to results presented by prior work, our solutions demonstrate significant improvements in throughput and frequency, thus validating the effectiveness of our workflow.

6.2 Future Directions

This project is the first step in creating a workflow to automate partitioned Systolic Array design with careful placement consideration. Beyond relying on Xilinx's architecture and software, we plan to extend this work to more platforms, and possible integration with Multi-Level Intermediate Representation (MLIR) [37].

6.2.1 Wider Platform Support

The current implementation focuses on Xilinx's Ultrascale+ platform, where the database generation and synthesis process rely on the Xilinx Vivado toolchain. However, Xilinx is not the only manufacturer of FPGA chips. Intel and Archonix are both FPGA companies working on the data center FPGA accelerator, where they have their own processing unit implementation. For example, Intel's latest Agilex FPGA has a configurable DSP engine with variable precision allowing them. Achronix's latest Speedster7t FPGA has machine learning processors (MLP) blocks constructed by a massively parallel array of computing elements with up to 32 multipliers operating up to 750 MHz. One promising direction is to extend the current code base to support different manufacturers' hardblocks and perform a performance analysis between the implementations.

6.2.2 MLIR Integration

The MLIR project provides a framework to construct reusable and extensible compiler infrastructure as an extension from the LLVM project [36]. TensorFlow [4] is an end-to-end open-source machine learning platform that uses MLIR as its foundation to build many of its utilities and tools. Integrating this optimizer with MLIR can further software abstraction for a more extensible and robust FPGA backend compiler. Instead of taking high-level topology of different CNNs, the optimizer can be extended to optimize for the *dialets* in MLIR, allowing more accurate designs.

References

[1] *Combinatorial Optimization: Theory and Algorithms*, pages 426–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi:10.1007/3-540-29297-7_18.

[2] After moore's law — technology quarterly, Feb 2016. URL: https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law.

[3] Introduction to cloud tpu, 2021. URL: https://cloud.google.com/tpu/docs/intro-to-tpu.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[5] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545, 2014. doi:10.1109/TASLP.2014.2339736.

[6] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, and Gerald Penn. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *2012 IEEE international conference on Acoustics, speech and signal processing (ICASSP)*, pages 4277–4280. IEEE, 2012.

[7] Paolo Arena, Adriano Basile, Maide Bucolo, and Luigi Fortuna. Image processing for medical diagnosis using cnn. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 497(1):174–178, 2003.

[8] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015. URL: http://iopscience.iop.org/article/10.1088/1749-4699/8/1/014008.

[9] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*, San Jose, CA, June 2013. USENIX Association. URL: https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/blott.

[10] Andreas Bortfeldt and Tobias Winter. A genetic algorithm for the two-dimensional knapsack problem with rectangular pieces. *Int. Trans. Oper. Res.*, 16:685–713, 2009.

[11] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 10–19, 2020. doi:10.1109/ICFPT51103.2020.00011.

[12] Long Chung Chan, G. Malik, and N. Kapre. Partitioning fpga-optimized systolic arrays for fun and profit. *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 144–152, 2019.

[13] TUL Corporation. Pynq™-z2 board product page. URL: https://www.tul.com.tw/ProductsPYNQ-Z2.html.

[14] L Nunes De Castro and Jon Timmis. An artificial immune network for multimodal function optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, volume 1, pages 699–704. IEEE, 2002.

[15] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug

Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th International Symposium on Computer Architecture, 2018*. ACM, June 2018. URL: https://www.microsoft.com/en-us/research/publication/a-configurable-cloud-scale-dnn-processor-for-real-time-ai/.

[16] Maayan Frid-Adar, Idit Diamant, Eyal Klang, Michal Amitai, Jacob Goldberger, and Hayit Greenspan. Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification. *Neurocomputing*, 321:321–331, 2018.

[17] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. Deep learning with int8 optimization on xilinx devices. *White Paper*, 2016.

[18] Adam Gaier and David Ha. Weight agnostic neural networks, 2019. arXiv:1906.04358.

[19] Francesco Gargiulo, Stefano Silvestri, and Mario Ciampi. Deep convolution neural network for extreme multi-label text classification. In *Healthinf*, pages 641–650, 2018.

[20] Ross Girshick. Fast r-cnn, 2015. arXiv:1504.08083.

[21] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2014. arXiv:1311.2524.

[22] David Ha. A visual guide to evolution strategies. *blog.otoro.net*, 2017. URL: https://blog.otoro.net/2017/10/29/visual-evolution-strategies/.

[23] Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[24] Nikolaus Hansen, yoshihikoueno, ARF1, Kento Nozawa, Matthew Chan, Youhei Akimoto, and Dimo Brockhoff. Cma-es/pycma: r3.1.0, June 2021. doi:10.5281/zenodo.5002422.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. arXiv:1512.03385.

[26] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. arXiv:1704.04861.

[27] Weilin Huang, Yu Qiao, and Xiaoou Tang. Robust scene text detection with convolution neural network induced mser trees. In *European conference on computer vision*, pages 497–511. Springer, 2014.

[28] Zhao Jianqiang, Gui Xiaolin, and Zhang Xuejun. Deep convolution neural networks for twitter sentiment analysis. *IEEE Access*, 6:23253–23260, 2018.

[29] Jukka Jylänki. A thousand ways to pack the bin-a practical approach to two-dimensional rectangle bin packing. 2010.

[30] Baris Kayalibay, Grady Jensen, and Patrick van der Smagt. Cnnbased segmentation of medical imaging data. *arXiv preprint arXiv:1701.03056*, 2017.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[32] Suhas Kumar. Fundamental limits to moore's law, 2015. arXiv:1511.05956.

[33] T Senthil Kumar. Video based traffic forecasting using convolution neural network model and transfer learning techniques. *Journal of Innovative Image Processing (JIIP)*, 2(03):128–134, 2020.

[34] H. T. Kung. Why systolic architectures? *Computer*, 15:37–46, 1982.

[35] Yan Lan, G. Dósa, X. Han, Chenyang Zhou, and A. Benko. 2d knapsack: Packing squares. *Theor. Comput. Sci.*, 508:35–40, 2013.

[36] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[37] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi:10.1109/CGO51591.2021.9370308.

[38] Shulin Li, Weigang Zhang, Guorong Li, Li Su, and Qingming Huang. Vehicle detection in uav traffic video based on convolution neural network. In *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, pages 1–6. IEEE, 2018.

[39] Hsien-I Lin, Ming-Hsiang Hsu, and Wei-Kai Chen. Human hand gesture recognition using a convolution neural network. In *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1038–1043. IEEE, 2014.

[40] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Jinwei Xu, Shijie Li, Yongmei Zhou, and Yingnan Xu. Throughput-optimized fpga accelerator for deep convolutional neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, 10(3), July 2017. doi:10.1145/3079758.

[41] Shih-Chung B. Lo, Heang-Ping Chan, Jyh-Shyan Lin, Huai Li, Matthew T. Freedman, and Seong K. Mun. Artificial convolution neural network for medical image pattern recognition. *Neural Networks*, 8(7):1201–1214, 1995. Automatic Target Recognition. URL: https://www.sciencedirect.com/science/article/pii/0893608095000615, doi:https://doi.org/10.1016/0893-6080(95)00061-5.

[42] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 13–24. IEEE Press, June 2014. Selected as an IEEE Micro TopPick. URL: https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/.

[43] V. Reddi, Christine Cheng, D. Kanter, P. Mattson, Guenther Schmuelling, Carole-Jean Wu, B. Anderson, Maximilien Breughe, Mark Charlebois, W. Chou, R. Chukka, Cody A. Coleman, S. Davis, P. Deng, Greg Diamos, J. Duke, D. Fick, J. Gardner, Itay Hubara, S. Idgunji, T. Jablin, Jeff Jiao, T. John, Pankaj Kanwar, D. Lee, Jeffery Liao, Anton Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, Gennady Pekhimenko, A. Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, M. Thomson, F. Wei, E. Wu, Lingjie Xu, K. Yamada, B. Yu, George Yuan, Aaron Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020.

[44] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016. arXiv:1506.02640.

[45] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016. arXiv:1506.01497.

[46] 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)*, pages 1–8, Sep. 2019.

[47] Ananda Samajdar, Yuhao Zhu, P. Whatmough, Matthew Mattina, and T. Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv: Distributed, Parallel, and Cluster Computing*, 2018.

[48] secnot. rectpack. https://github.com/secnot/rectpack, 2018.

[49] 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)*, pages 1–4, Aug 2016. doi:10.1109/FPL.2016.7577315.

[50] Patrick Siarry, Alain Pétrowski, and Mourad Bessaou. A multipopulation genetic algorithm aimed at multimodal optimization. *Advances in Engineering Software*, 33(4):207–213, 2002.

[51] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[52] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014. arXiv:1409.4842.

[53] Jasper Uijlings, K. Sande, T. Gevers, and A.W.M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104:154–171, 09 2013. doi:10.1007/s11263-013-0620-5.

[54] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 93–104, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3431920.3439292.

[55] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 3381–3387. IEEE, 2008.

[56] Ephrem Wu, Xiaoqian Zhang, David Berman, Inkeun Cho, and John 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*, pages 191–200, 2019. doi:10.1145/3289602.3293925.

[57] Xilinx. Ultrascale architecture and product data sheet. URL: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[58] Xilinx. Ultrascale architecture dsp slice user guide. URL: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[59] Xilinx. Ultrascale architecture memory resources. URL: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.

[60] Xilinx. Virtex ultrascale+ fpga data sheet: Dc and ac switching characteristics. URL: https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf.

[61] Xilinx. Xilinx large fpga methodology guide. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf.

[62] Xilinx. Xilinx ai engines and their applications. Technical report, July 2020. URL: https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf.

[63] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. Autodnnchip. *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb 2020. URL: http://dx.doi.org/10.1145/3373087.3375306, doi:10.1145/3373087.3375306.

[64] Kang Zhang, Shengchang Lan, and Guiyuan Zhang. On the effect of training convolution neural network for millimeter-wave radar-based hand gesture recognition. *Sensors*, 21(1), 2021. URL: https://www.mdpi.com/1424-8220/21/1/259, doi:10.3390/s21010259.

[65] Y. Zhao, X. Gao, X. Guo, J. Liu, E. Wang, R. Mullins, P. Y. K. Cheung, G. Constantinides, and C. Xu. Automatic generation of multi-precision multi-arithmetic

cnn accelerators for fpgas. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 45–53, 2019. doi:10.1109/ICFPT47387.2019.00014.

[66] Li Zheng, Qiao Li, Hua Ban, and Shuhua Liu. Speech emotion recognition based on convolution neural network combined with random forest. In *2018 Chinese Control And Decision Conference (CCDC)*, pages 4143–4147, 2018. doi:10.1109/CCDC.2018.8407844.

APPENDICES

Appendix A

RECTPACK profiles example and guide

Each profile must provide two attribute: 1) `width_height_multiplier` and 2) `DSP_dimensions`. The `width_height_multiplier` represents the ratio of how many input is packed in each row versus each column. In our case, we enable double packing on each row, and the ratio we use is 2 : 1. The `DSP_dimensions` contains the list of dimensions for each bin in the KnapSack model. In Listing A.1, we consider each SLRs as their own bin. For VU37P, there are 94 DSP48E2 hard blocks in each column, and each SLR contains 32 columns. For VU9P, there are 120 DSP48E2 hard blocks in each column, and each SLR contains 19 columns. This level of abstraction allows us to create more profiles targeting specific time regions on the FPGA die.

```
xcvu37p-full: # Name of the profile
    # 2 for double packing in the horizontal axis
    width_height_multiplier: [2, 1]
    DSP_diemsnsions: [[32, 94], [32, 94], [32, 94]]

xcvu9p-full: # Name of the profile
    # 2 for double packing in the horizontal axis
    width_height_multiplier: [2, 1]
    DSP_diemsnsions: [[19, 120], [19, 120], [19, 120]]
```

Listing A.1: KnapSack profiles example on VU37P and VU9P

Appendix B

Optimizer configuration file detail

This configuration file is a general configuration that contains six different sections of parameters:

1. OPTIMIZER - configure how the many resources the optimizer will use
2. PATH - configure the path to all the datasets and results
3. CNN - configure the convolution neural network-related parameters
4. FPGA - configure the max amount of processing elements available
5. CMA-ES - configure parameters for CMA-ES and the parameters for frequency setting
6. RECTPACK - configure which profile will be used and whether the placement restriction will be respected

OPTIMIZER This section contains two parameters: 1) the number of trials an optimizer can attempt when the optimizer discovers no good design and 2) the number of threads allowed.

PATH This section contains four path parameters:

1. `dump_path` - location of the resulting CSV

2. `topologies_path` - location of CSV files describing topologies of CNNs
3. `freq_data_path` - location of the CSV files containing the frequency dataset - $freq[][]$
4. `cycle_data_path` - location of the CSV files containing the cycle dataset - $cyc[][]$

CNN This section contains a list of information regarding the target of each neural network. For each network target, three information are needed:

1. name of the neural network
2. the number of layers in the neural network
3. the maximum group count - k that optimizer will split the network workload into

The value of k is usually to set half of the number of layers in the network. When the value of k is equal to the number of layers, each layer gets its grouping. From our experiments, the optimal k value is found when it is equal to or less than half of the layer in the network.

FPGA This section contains only one parameter - the number of available processing elements on the FPGA die. When using all 3 SLRs on VU37P and VU9P, the value should be set to $17280 = (32 * 2) * 90 * 3$ and $13680 = (19 * 2) * 120 * 3$ respectively. In cases where specific clock regions need to be focused, we can adjust the value of this parameter accordingly. The configuration file accepts a list of available processing elements count, and the optimizer parser will automatically match the correct count value to the available KnapSack profile.

CMA-ES This section contains a list of parameters to configure the CMA-ES model, including:

1. Population Size (pop_size) - Number of off-springs sampled in each iteration
2. Maximum Iteration (max_iteration) - Number of iterations the CMA-ES can last
3. Sigma (sigma) - Value of the sigma for the initial normal distribution used in CMA-ES
4. Penalty Offset (penalty_offset) - Value of penalty value used in the cost function

5. Processing elements steps (pe_step) - Size of the steps used in the resource assignment process

On top of configuration for CMA-ES, the frequency optimization setting is also included in this section. There are three major settings, 1) `enable_frequency_score`, 2) `frequency_type` and 3) `fall_back_frequency`. When frequency optimization is enabled and `frequency_type` equals to `unbounded`, the optimizer will generate designs in the **Unbounded Frequency** variation. When frequency optimization is enabled and `frequency_type` equals to `bounded`, the optimizer will generate designs in the **Bounded Frequency** variation. `fall_back_frequency` is used whenever a frequency needed does not exist in the frequency database and no estimation can be provided.

RECKPACK This section contains a list of KnapSack profiles and all the packing strategies the optimizer will go through. The two packing strategies being supported right now are `unconstrained` leading to the **Unconstrained Partition Sizing** variant; and `placement_aware` leading to both **Packing-Aware** variant.

The parser use both the configuration files to generate a set of **flavours**. The set of **flavours** is constructed by having all the combinations of:

1. Each networks
2. Each unique valid k value
3. Each possible frequency setting
4. Each packing strategies
5. Each KnapSack profiles

The optimizer then spawns multiple processing instances for each thread available. This set of **flavors** is then passed to each processing instance.

Appendix C

Systolic Array Codebase Overview

In Figure C.1, we represent an overview of how the codebase works together. All the components that are platform-specific are highlighted in red.

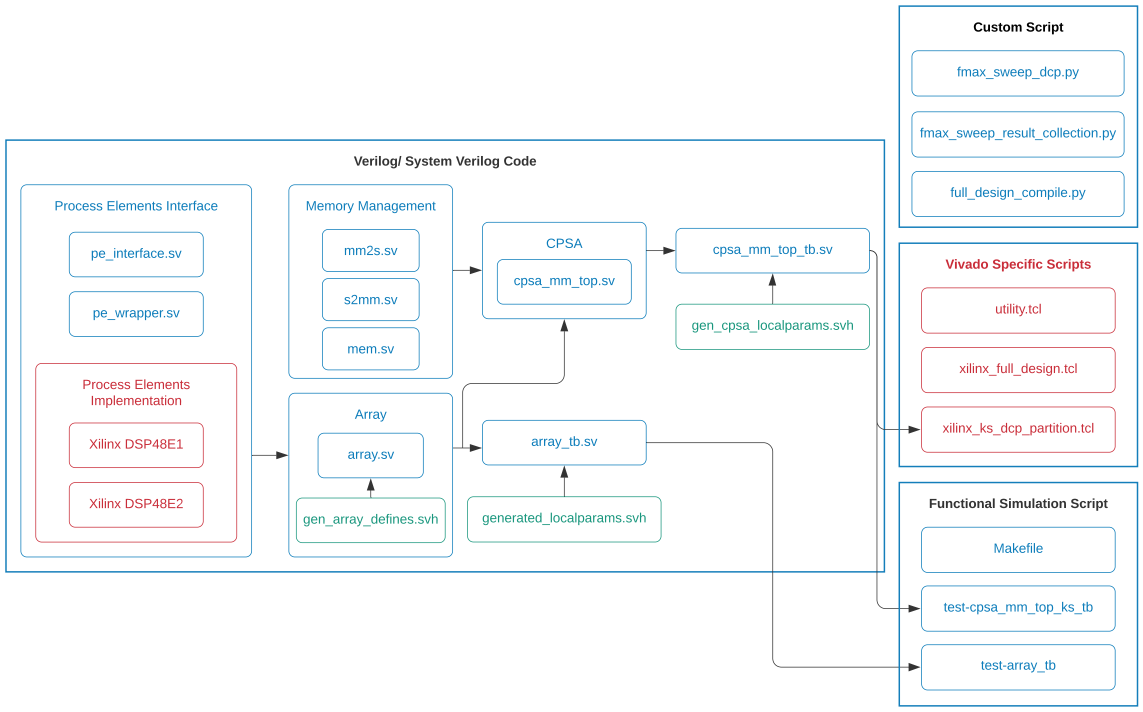


Figure C.1: An overview of the systolic array code base.

The *Verilog/System Verilog Code* module includes all the hardware codes used to construct the systolic array. We abstract the usage of processing elements with a public interface in `pe_interface.sv`. This allows us to define the target function of the processing element easily. `array.sv` describes how each processing element are connected to form a computation unit and can refer to a specific processing element implementation via `gen_array_defines.svh`. To achieve a fully functional design, the final `cpsa_mm_top.sv` module contains multiple memory module `mm2s.sv`, `s2mm.sv` and `mem.sv` together with the `array.sv`.

In this repository, we also provide scripts allowing the user to easily configure the dimension of a systolic array and the target matrix to be computed in the `Makefile`. User can choose to functional verify only the array via `test-array_tb` text case or the full implementation with memory control via `test-cpsa_mm_top_ks_tb` text case.

This repository also contains all the custom scripts that we use to create the frequency model. `fmax_sweep_dcp.py` creates all the necessary files, including the placement constraint files and Verilog code to sweep over systolic arrays with different dimensions across the staggered location. `fmax_sweep_result_collection.py` extracts all the timing info and reorganizes all the results back to a comma-separated values (CSV) file for the optimizer. After the optimizer proposes an optimized design in the next stage, `full_design_compile.py` will search for the correct DCP files and recompile the final design.

Appendix D

YAML configuration file targeting VU9P

D.1 From the Command Line

Running `python3 ks_optimizer.py --config vu37p.yaml --execute True` will automatically set up all the directories and start the optimization process.

Running `python3 ks_optimizer.py --config vu37p.yaml --execute False` will automatically set up all the directories and provide basic statistic on the quantity of flavours that will be optimized.

The definition of the RECTPACK profiles use in the Listing D.1 are provided in the Listing D.2.

```yaml
# Optimizer Parameters
OPTI:
    max_trials: 10
    threads: 12

# Path Parameters
PATH:
    dump_path: 'results/VU9P'
    topologies_path: 'database/topologies/'
    freq_data_path: 'database/frequency_data/vu9p/'
    cycle_data_path: 'database/cycle_data/'
```

```

# DNN Parameters
DNN:
    nets: [
        ['FasterRCNN', 46, 23], # NameOfDNN, NumOfLayers
  , MaxK
        ['mobilenet', 27, 27],
        ['yolo_tiny', 10, 9],
        ['googlenet', 58, 29],
        ['alexnet', 8, 8],
        ['AlphaGoZero', 8, 8],
        ['ncf_rec', 8, 8],
        ['resnet_50_v1', 53, 27]
    ]

# FPGA Parameters
FPGA:
    max_pe_units: [
        12960,  # 18*2 * 120 * 3
        4320,   # 18*2 * 120
        2400,   # 10*2 * 120
        1920,   # 8*2 * 120
        2160    # 18*2 * 60
    ]

# CMA-es Parameters
CMA-ES:
    target_cols: [
        'DRAM_cycle'
    ]
    seedings: [
        'optimised'
    ]
    pop_size: [100]
    max_iteration: [10000]
    sigma: [0.5]
    penalty_offset: [100000000]
    pe_step: [2]

    # CMA-es Parameters for frequency optimiztion
    FREQUENCY_SETTING:
        enable_frequency_score: [
            True,
            False
        ]
        # this setting is only applied if
  enable_frequency_score = True
        frequency_type: [
            'unbounded',
            'bounded'
        ]
        fall_back_frequency: [
            500
        ]

# RectPack Parameters
RECTPACK:
    rectpack_configs: [
        'xcvu9p-full',
        'xcvu9p-3-times',
        'xcvu9p-6-times-x-l',
        'xcvu9p-6-times-x-r',
        'xcvu9p-6-times-y'
    ]
    packing_strategies: [
        'unconstrained',
        'dsp_placement_aware_sizing'
    ]
```

Listing D.1: Configuration profiles on the VU9P run

```
# Considering all 3 SLRs region
xcvu9p-full:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[19, 120], [19, 120], [19, 120]]

# Considering 1 SLR region
```

```
xcvu9p-3-times:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[19, 120]]

# Considering 1/2 SLR (half in x-axis, X0 to X2)
xcvu9p-6-times-x-l:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[11, 120]]

# Considering 1/2 SLR (half in x-axis, X3 to X5)
xcvu9p-6-times-x-r:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[8, 120]]

# Considering 1/2 SLR (half in y-axis)
xcvu9p-6-times-y:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[18, 60]]
```

Listing D.2: KnapSack profiles modelling VU9P

Appendix E

YAML configuration file targeting VU37P

E.1 From the Command Line

Running `python3 ks_optimizer.py --config vu9p.yaml --execute True` will automatically set up all the directories and start the optimization process.

Running `python3 ks_optimizer.py --config vu9p.yaml --execute False` will automatically set up all the directories and provide basic statistic on the quantity of flavours that will be optimized.

The definition of the RECTPACK profiles use in the Listing E.1 are provided in the Listing E.2.

```yaml
# Optimizer Parameters
OPTI:
    max_trials: 10
    threads: 12

# Path Parameters
PATH:
    dump_path: 'results/VU37P'
    topologies_path: 'database/topologies/'
    freq_data_path: 'database/frequency_data/vu37p/'
    cycle_data_path: 'database/cycle_data/'
```

```

# DNN Parameters
DNN:
    nets: [
        ['FasterRCNN', 46, 23], # NameOfDNN, NumOfLayers
    , MaxK
        ['mobilenet', 27, 27],
        ['yolo_tiny', 10, 9],
        ['googlenet', 58, 29],
        ['alexnet', 8, 8],
        ['AlphaGoZero', 8, 8],
        ['ncf_rec', 8, 8],
        ['resnet_50_v1', 53, 27]
    ]

# FPGA Parameters
FPGA:
    # We pick a maximum of 17280 systolic units for
  consistency with Chan et. al.
    max_pe_units: [
        17280,  # 32*2 * 90 * 3
        5760,   # 32*2 * 90
        2880,   # 16*2 * 90
        1408,   # 16*2 * 44
        2816    # 32*2 * 44
    ]

# CMA-es Parameters
CMA-ES:
    target_cols: [
        'DRAM_cycle'
    ]
    seedings: [
        'optimised'
    ]
    pop_size: [100]
    max_iteration: [10000]
    sigma: [0.5]
    penalty_offset: [100000000]
    pe_step: [2]

    # CMA-es Parameters for frequency optimiztion
    FREQUENCY_SETTING:
        enable_frequency_score: [
            True,
            False
        ]
        # this setting is only applied if
  enable_frequency_score = True
        frequency_type: [
            'unbounded',
            'bounded'
        ]
        fall_back_frequency: [
            500
        ]

# RectPack Parameters
RECTPACK:
    rectpack_configs: [
        'xcvu37p-full',
        'xcvu37p-3-times',
        'xcvu37p-6-times-x',
        'xcvu37p-6-times-y',
        'xcvu37p-12-times-x-y',
    ]
    packing_strategies: [
        'unconstrained',
        'dsp_placement_aware_sizing'
    ]
```

Listing E.1: Configuration profiles on the VU37P run

```
# Considering all 3 SLRs region
xcvu37p-full:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[32, 90], [32, 90], [32, 90]]
```

```

# Considering 1 SLR region
xcvu37p-3-times:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[32, 90]]

# Considering 1/2 SLR (half in x-axis)
xcvu37p-6-times-x:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[16, 90]]

# Considering 1/2 SLR (half in y-axis)
xcvu37p-6-times-y:
    width_height_multiplier: [2, 1]
    DSP_diemsnsion: [[32, 45]]
```
Listing E.2: KnapSack profiles modelling VU37P