RapidLayout: Fast Hard Block Placement of FPGA-optimized Systolic Arrays using Evolutionary Algorithm

NIANSONG ZHANG, Sun Yat-sen University, China
XIANG CHEN, Sun Yat-sen University, China
NACHIKET KAPRE, University of Waterloo, Canada

Evolutionary algorithms can outperform conventional placement algorithms such as simulated annealing, analytical placement, and manual placement on runtime, wirelength, pipelining cost, and clock frequency when mapping hard block intensive designs such as systolic arrays on Xilinx UltraScale+ FPGAs. For certain hard-block intensive designs, the commercial-grade Xilinx Vivado CAD tool cannot provide legal routing solutions without tedious manual placement constraints. Instead, we formulate hard block placement as a multi-objective optimization problem that targets wirelength squared and bounding box size. We build an end-to-end placement-and-routing flow called RapidLayout using the Xilinx RapidWright framework. RapidLayout runs 5–6× faster than Vivado with manual constraints and eliminates the weeks-long effort to manually generate placement constraints. RapidLayout enables transfer learning from similar devices and bootstrapping from much smaller devices. Transfer learning in the UltraScale+ family achieves 11–14× shorter runtime, and bootstrapping from a 97% smaller device delivers 2.1–3.2× faster optimizations. RapidLayout outperforms (1) a tuned simulated annealer by 2.7–30.8× in runtime while achieving similar quality of results, (2) VPR by 1.5× in runtime, 1.9–2.4× in wirelength, and 3–4× in bounding box size, while also (3) beating the analytical placer UTPlaceF by 9.3× in runtime, 1.8–2.2× in wirelength, and 2–2.7× in bounding box size.

CCS Concepts: • **Hardware → Partitioning and floorplanning**; **Placement**.

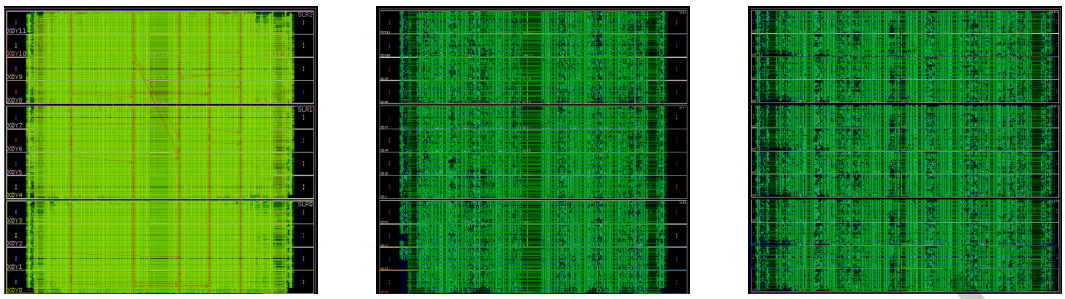
Additional Key Words and Phrases: FPGA Placement, Systolic Array, Evolutionary Algorithm

1 INTRODUCTION

Modern high-end FPGAs provide high compute density with a heterogeneous mixture of millions of classic lookup tables and programmable routing networks along with tens of thousands of DSP and RAM hard blocks. These hard blocks offer ASIC-like density and performance for signal processing functions and on-chip SRAM access. For example, Xilinx UltraScale+ VU11P is equipped with 960 UltraRAM blocks, 4032 Block RAM slices, and 9216 DSP48 blocks capable of operating at 650–891 MHz frequencies which are typically unheard of with LUT-only designs. Furthermore, these hard blocks provide specialized nearest-neighbor interconnect for high-bandwidth,

This work is supported by the National Key Research and Development Program of China (No. 2019YFE0196400), Guangdong R&D Project in Key Areas under Grant (2019B010158001,2019B010156004), Industry-University-Research Cooperation Project in Zhuhai (No. ZH22017001200072PWC), and Industry-University Collaborative Education Program between SYSU and Digilent Technology: Edge AI Oriented Open Source Software and Hardware Makerspace. This work is also supported in part by MITACS Globalink Research Internship.
Authors' addresses: Niansong Zhang, ,~zhangns@mail2.sysu.edu.cn, Sun Yat-sen University, 132 Outer Ring East Road, Guangzhou, Guangdong, China, 510006; Xiang Chen, (*correspondingauthor),~chenxiang@mail.sysu.edu.cn, Sun Yat-sen University, 132 Outer Ring East Road, Guangzhou, Guangdong, China, 510006; Nachiket Kapre, ,~nachiket@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L3G1.

1936-7406/2022/2-ART $15.00
https://doi.org/10.1145/3501803



(a) Unroutable floorplan in Vivado with heavy congestion

(b) Manual constraints in Vivado routes in 5–6 hours.

(c) RapidLayout automatic placement routes in ≈1 hour.

Fig. 1. Floorplanning scenarios for a neural network accelerator with 480 convolution blocks mapped to Xilinx UltraScale+ VU11P. Vivado cannot provide a routable floorplan without manually generated placement constraints, while RapidLayout automatically discovers a high-quality floorplan and runs 5–6× faster.

low-latency *cascade* data movement. These features make it particularly attractive for building systolic neural network accelerators such as CLP [42, 43], Cascades [40], and Xilinx SuperTile [49, 50].

Exploiting the full capacity of FPGA resources including hard blocks at high clock frequency is challenging. The CLP designs presented in [42, 43] only operate at 100–170 MHz on Virtex-7 FPGAs but leave DSPs unused. The Xilinx SuperTile [49, 50] designs run at 720 MHz, but leave half of the DSPs unused, and also waste URAM bandwidth by limiting access. The chip-spanning 650 MHz 1920×9 systolic array design for the VU11P FPGA [40] requires 95% or more of the hard block resources but fails to route in commercial-grade Xilinx Vivado run with high effort due to congestion. Manual placement constraints are necessary to enable successful bitstream generation, but this requires weeks of painful trial-and-error effort and visual cues in the Vivado floorplanner for the correct setup. This effort is needed largely due to the irregularity and asymmetry of the columnar DSP and RAM fabric and the complex cascade constraints that must be obeyed for the systolic data movement architecture. Once the constraints are configured, Vivado still needs 5–6 hours of compilation time, making design iteration long and inefficient. Furthermore, to ensure high-frequency operation, it becomes necessary to pipeline long wires in the design. Since timing analysis must be done post-implementation, we end up either suffering the long CAD iteration cycles or overprovisioning unnecessary pipelining registers to avoid the long design times.

Given this state of affairs with the existing tools, we develop RapidLayout: an alternative, automated, fast placement approach for hard block designs. We compare the routing congestion maps produced by RapidLayout with Vivado in Fig. 1 and note the ability to match low-congestion manual placement effort. It is important that such a toolflow addresses the shortcomings of the manual approach by (1) discovering correct placements quickly without the manual trial-and-error loop through slow Vivado invocations, (2) encoding the complex placement restrictions of the data movement within the systolic architecture in the automated algorithm, (3) providing fast wirelength estimation to permit rapid objective function evaluation of candidate solutions, and (4) exploiting design symmetry and overcoming irregularity of the columnar FPGA hard block architecture. Given this wish list, we used the Xilinx RapidWright framework for our tool.

At its core, the toolflow is organized around the design of a novel evolutionary algorithm formulation for hard block placement on the FPGA through multi-objective optimization of wirelength squared and bounding box metrics. Given the rapid progress in machine learning tools, there is an opportunity to revisit conventional CAD algorithms [9], including those in this paper, and attack them with this new toolbox.

The key contributions of this work are listed as follows:

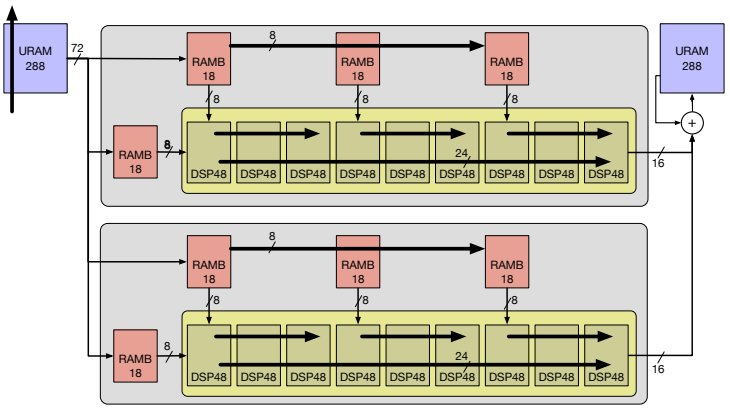


Fig. 2. Convolutional Building Block for FPGA-Optimized Systolic Array in [40]. Cascade URAM, BRAM, and DSP links are highlighted in bold.

- We formulate a novel FPGA placement problem for tens of thousands of hard blocks as a multi-objective optimization using evolutionary techniques.
- We quantify quality of result (QoR) metrics including runtime, wirelength, bounding box size, clock frequency, and pipelining cost for the evolutionary placement algorithms NSGA-II and CMA-ES. We compare these metrics against conventional Simulated Annealing (SA), Genetic Algorithm (GA), Versatile-Place-and-Route (VPR) [31], and the state-of-art analytical placer UTPlaceF [28].
- We build an end-to-end RapidLayout placement-and-routing toolflow using the open-source Xilinx RapidWright framework.
- We develop the transfer learning and bootstrap process to further accelerate placement optimization. Transfer learning migrates existing hard block placement from base devices to similar devices in the UltraScale+ family (VU3P–VU13P). Placement bootstrap extrapolates placement from much smaller base devices to obtain high-quality initialization on target devices.

2 BACKGROUND

We first discuss the hard block intensive systolic array accelerator optimized for the Xilinx UltraScale+ FPGAs. Next, we discuss the Xilinx RapidWright framework for programming FPGAs through a non-RTL design flow. Then, we describe previous research on FPGA placement algorithms. After that, we review the classic NSGA-II algorithm and the state-of-art CMA-ES algorithm and compare them with previous evolutionary placement efforts. Finally, we review the transfer learning applications in machine learning and EDA domains.

2.1 FPGA-optimized Systolic Array Accelerator

Systolic arrays [22, 25] are tailor-made for convolution and matrix operations needed for neural network acceleration. They are constructed to support extensive data reuse through nearest-neighbor wiring between a simple 2D array of multiply-accumulate blocks. They are particularly amenable to implementation on the Xilinx UltraScale+ architecture with cascade nearest-neighbor connections between DSP, BRAM, and URAM hard blocks. We utilize the systolic convolutional neural network accelerator presented in [40] and illustrated in Fig. 2. The key repeating computational block is a convolution engine optimized for the commonly-used 3×3 convolution operation. This is implemented across a chain of 9 DSP48 blocks by cascading the accumulators. Furthermore, row reuse is supported by cascading three BRAMs to supply data to a set of three DSP48s each.

Finally, the URAMs are cascaded to exploit all-to-all reuse between the input and output channels in one neural network layer. Overall, when replicated to span the entire FPGA, this architecture uses 95–100% of the DSP, BRAM, and URAM resources of the high-end UltraScale+ VU37P device. When mapped directly using Vivado without any placement constraints, the router runs out of wiring capacity to fit the connections between these blocks. Since the convolution block is replicated multiple times to generate the complete accelerator, it may appear that placement should be straightforward. However, due to irregular interleaving of the hard block columns, and the non-uniform distribution of resources, the placement required to fit the design is quite tedious and takes weeks of effort.

2.2 RapidWright

In this paper, we develop our tool based on the Xilinx RapidWright [26] open-source FPGA framework. It aims to improve FPGA designers' productivity and design QoR by composing large FPGA designs through a pre-implemented and modular methodology. RapidWright provides high-level Java API access to low-level Xilinx device resources. It supports design generation, placement, routing, and allows design checkpoint (DCP) integration for seamless inter-operability with the Xilinx Vivado CAD tool to support custom flows. It also provides access to device geometry information that enables wirelength calculations crucial for tools that aim to optimize timing.

2.3 FPGA Placement

FPGA placement maps a clustered logical circuit to an array of fixed physical components to optimize routing area, critical path, power efficiency, and other metrics. FPGA placement algorithms can be broadly classified into four categories: (1) classic min-cut partitioning [32, 33, 46], (2) popular simulated-annealing-based methods [2, 3, 23, 31], (3) analytical placement currently used in FPGA CAD tools [1, 12, 15, 28], and (4) esoteric evolutionary approaches [7, 20, 47]. Min-cut algorithm worked well on small FPGA capacities by iteratively partitioning the circuit to spread the cells across the device. Simulated Annealing was the popular choice for placement until recently. It operates by randomly swapping clusters in an iterative, temperature-controlled fashion resulting in progressively higher quality results. Analytical placers are currently industry standard as they solve the placement problem using a linear algebraic approach that delivers higher quality solutions with less time than annealing. For example, Vivado uses an analytical placement to optimize timing, congestion, and wirelength [12].

2.4 Evolutionary Algorithms

There have been several attempts to deploy evolutionary algorithms for FPGA placement with limited success. The earliest one by Venkatraman and Patnaik [47] encodes each two-dimensional block location in a gene and evaluates the population with a fitness function for critical path and area efficiency. More recently, P. Jamieson [18], [19] points out that GAs for FPGA placement are inferior to annealing mainly due to the crossover operator's weakness and proposed a clustering technique called supergenes [20] to improve its performance.

In this paper, we design a novel combinational gene representation for FPGA hard block placement and explore two evolutionary algorithms:

1. **NSGA-II**: Non-Dominated Sorting Genetic Algorithm (NSGA-II [10]) is a two-decade-old multi-objective evolutionary algorithm that has grown in popularity today for Deep Reinforcement Learning [27] and Neural Architecture Search [29] applications. NSGA-II addresses multi-objective selection with non-dominated filtering and crowd distance sorting, which allow the algorithm to effectively explore the solution space and preserve good candidates.
2. **CMA-ES**: Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) is a continuous domain optimization algorithm for non-linear, ill-conditioned, black-box problems [16]. CMA-ES models candidate solutions as

samplings of an n-dimensional Gaussian variable with mean μ and covariance matrix $\mathbf{C}_\sigma$. At each evolutionary iteration, the population is generated by sampling from $\mathbb{R}^n$ with updated mean and covariance matrix. Here, crossover and mutation become adding Gaussian noise to the samplings, which overcomes the weakness of GA's crossover operator. We use the high-dimensional variant proposed in [37] for fast operation in our placement challenge.

2.5 Transfer Learning

Transfer learning is the practice of using existing models or knowledge as a starting point for new tasks or domains [35]. Transfer learning is commonly used in many fields, such as Computer Vision [14, 24, 36, 57], Natural Language Processing [5, 17, 38], Reinforcement Learning [8, 41, 45], and Multi-task Learning [21, 53, 56], to adapt to new data, tasks, or domains, and to accelerate the learning process. With the recent effort to solve EDA problems with machine learning methods, transfer learning is also used in the FPGA routing problem. RouteNet [39] proposed using Graph Neural Networks (GNN) to predict routing delay and jitter, and its jitter model was bootstrapped from the trained delay model. In Painting on Placement [55], transfer learning was used to improve the robustness of its routing congestion prediction model. We propose a conceptually adjacent idea for FPGA placement: transferring hard block placement from base devices with similar or much fewer amount of resources to accelerate placement optimization and improve QoR. To the best of our knowledge, this is the first work to discuss transfer learning for FPGA placement.

3 RAPIDLAYOUT

The challenge for mapping FPGA-optimized systolic arrays to the Xilinx UltraScale+ device is the placement of hard blocks to their non-uniform, irregular, columnar locations on the fabric while obeying the cascade data movement constraints. We first present our problem formulation and then discuss how to embed it into the evolutionary algorithms.

3.1 Problem Formulation

To tackle the placement challenge, we formulate the coarse-grained placement of RAMs and DSP blocks as a constrained multi-objective optimization problem. The placement for the rest of the logic *i.e.* lookup tables (LUTs) and flip-flops (FFs) is left to Vivado's placer. The multi-objective optimization goal is formalized as follows.

$$min \sum_{i,j} ((\Delta x_{i,j} + \Delta y_{i,j}) \cdot w_{i,j})^2 \tag{1}$$

$$\min(\max_k BBoxSize(C_k)) \tag{2}$$

subject to:

$$0 \leq x_i, y_i < XMAX, YMAX \tag{3}$$

$$x_i, y_i \neq x_j, y_j \tag{4}$$

if i is cascaded after j in the same column: $x_i = x_j$

$$y_i = \begin{cases} y_j + 1 & i,j \in \{DSP, URAM\} \\ y_j + 2 & i,j \in \{RAMB\} \end{cases} \tag{5}$$

In the equations above:

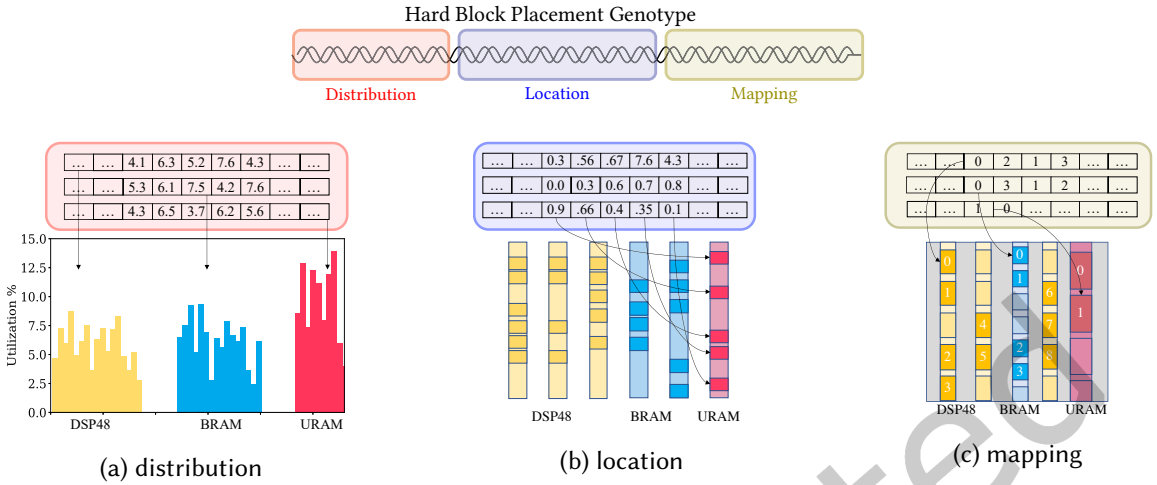


Fig. 3. Our three-tier genotype design for hard block placement. (a) Distribution defines the number of hard blocks to be placed in each column. (b) Location encodes the relative position of the corresponding hard blocks in its column. (c) Mapping defines the connectivity of hard blocks, i.e., which hard blocks are mapped to one convolution unit. The selected physical hard-block groups are numbered, which corresponds to the mapping genotype.

- $i \in \{DSP, RAM, URAM\}$ denotes a physical hard block to which a logic block is mapped.
- C_k denotes a convolution unit k that contains 2 URAMs, 18 DSPs, and 8 BRAMs.
- $\Delta x_{i,j} + \Delta y_{i,j}$ is the Manhattan distance between two physical hard blocks i and j.
- $w_{i,j}$ is the weight for wirelength estimation. Here we use the number of connections between hard blocks i and j.
- $BBoxSize()$ is the bounding box rectangle size (width + height) containing the hard blocks of a convolution unit C_k.
- x_i and y_i denote the RPM absolute grid co-ordinates of hard block i that are needed to compute wirelength and bounding box sizes [52].

Understanding the Objective Function: We approximate routing congestion performance with squared wirelength (Equation 1) and critical path length with the maximum bounding box size (Equation 2). These twin objectives try to reduce pipelining requirements while maximizing clock frequency of operation. While these optimization targets may seem odd, we have observed cases where chasing wirelength^2 alone has misled the optimizer into generating wide bounding boxes for a few stray convolution blocks. In contrast, optimizing for maximum bounding box alone was observed to be extremely unstable and causing convergence problems. Hence, we choose these two objective functions to restrict the spread of programmable fabric routing resources and reduce the length of the critical path between hard blocks and associated control logic fanout.

Understanding Constraints The optimizer only needs to obey three constraints. The **region constraint** in Equation 3 restricts the set of legal locations for the hard blocks to a particular repeatable rectangular region of size XMAX×YMAX on the FPGA. The **exclusivity constraint** in Equation 4 forces the optimizer to prevent multiple hard blocks from being assigned to the same physical location. The **cascade constraint** in Equation 5 is the "uphill" connectivity restriction imposed due to the nature of the Xilinx UltraScale+ DSP, BRAM, and URAM cascades. For DSPs and URAMs, it is sufficient to place connected blocks next to each other. For BRAMs, the adjacent block of the same type resides one block away from the current location. This is because RAMB180 and RAMB181, which are both RAMB18 blocks, are interleaved in the same column.

3.1.1 Genotype Design for Evolutionary Algorithms. We decompose placement into three sub-problems and encode the candidate solutions with a novel composite genotype design.

1. **Distribution** Since the systolic array accelerator does not match the hard block resource capacity perfectly, we allocate hard blocks across resource columns according to the distribution genotype.
2. **Location** Once we choose the exact number of blocks to place on a given resource column, we assign each block in the column a location according to a value between $0 \rightarrow 1$.
3. **Mapping** Finally, we label selected blocks and allocate them to each convolution unit according to the mapping genotype. It is a permutation genotype that optimizes the order of elements without changing their values.

In Fig. 3, we visualize the genotype design which consists of the three parts just discussed earlier. During evolution, each part of the genotype is updated and decoded independently, but evaluated together.

3.1.2 The Generality of the Evolutionary Formulation. The problem formulation and genotype design are motivated by a convolutional systolic array. However, the placement formulation is not limited to any particular hard block design. For example, the computing unit C_k (Equation 2) can be any hard block design, as long as the number of hard blocks, their connections, and cascade information are provided.

3.1.3 Comparison with Prior Evolutionary Placements. Our genotype design differs from prior works in three aspects: (1) We provide placement support for heterogeneous hard block types. (2) We encode cascade constraints into the genotype, which eschews extra legalization steps and reduces search space. (3) The three-tier genotype design enables placement transfer learning across devices (Section 4.5).

3.2 RapidLayout Design Flow

We now describe the end-to-end RapidLayout design flow: In Fig. 4, we illustrate the different stages of the flow, the approximate runtime of each stage, its interaction with RapidWright and Vivado. We use the convolutional systolic array design described in Section 2.1 as an example, and the target device is Xilinx UltraScale+ VU11P. The different stages of the tool are described below:

- **Netlist Replication (<1 s)** RapidLayout starts with a synthesized netlist of the convolution unit with direct instantiations of the FPGA hard blocks. The unit design is replicated into the entire logical netlist that maps to the whole Super Logic Region (SLR).
- **Evolutionary Hard Block Placement (30 s-5min)** RapidLayout uses NSGA-II or CMA-ES to generate hard block placement for the minimum repeating rectangular region. Then the rectangular layout is replicated (*copy-paste*) to produce the placement solution for the entire SLR.
- **Placement and Site Routing (≈3 min)** The placement information is embedded in the DCP netlist by placing the hard blocks on the physical blocks called "sites", followed by "site-routing" to connect intra-site pins.
- **Post-Placement Pipelining (≈10 s)** After finalizing placement, we can compute the wirelength for each net in the design and determine the amount of pipelining required for high-frequency operation. This is done post-placement [11, 44, 48] to ensure the correct nets are pipelined and to the right extent. The objective of this step is to aim for the 650 MHz URAM-limited operation as dictated by the architectural constraints of the systolic array [40].
- **SLR Placement and Routing (≈55 min)** Once the hard blocks are placed and pipelined, we call Vivado to complete LUT/FF placement and routing for the SLR.
- **SLR Replication (1-5 min)** The routed design on the SLR is copied across the entire device using RapidWright APIs to complete the overall implementation.

For a VU11P device, RapidLayout accelerates the end-to-end implementation by ≈5–6× when measuring CAD runtime alone (≈one hour vs. Vivado's 5–6 hours). This does not include the weeks of manual tuning effort that is avoided by automatically discovering the best placement for the design.

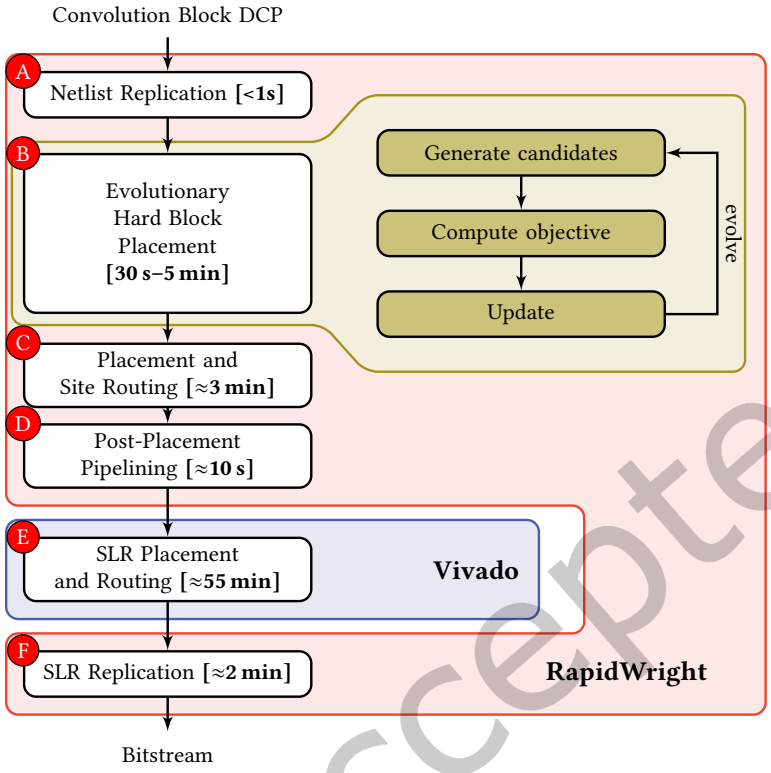


Fig. 4. RapidLayout Design Flow with runtime details for the Xilinx VU11P FPGA along with tool usage information. A bulk of the intelligent exploration is done in RapidWright, and Vivado is only invoked at the end for final placement and routing.

3.3 Full-chip Layout Example Walkthrough

To illustrate how the different steps help produce the full-chip layout, we walk you through the intermediate stages of the flow. We will inspect three stages of placement, going from a single block layout to a repeating rectangle layout, and then onto a full-chip layout.

Single Block layout: The floorplan of a single convolution block in isolation is shown in Fig. 5, where the hard block columns and chosen resources are highlighted. We also highlight the extent of routing requirements between the hard blocks in gray. The locations of the URAM, BRAM, and DSP columns are irregular, which forces a particular arrangement and selection of resources to minimize wirelength and bounding box size. It is clear that a simple *copy-paste* of a single block is not workable due to this irregularity.

Single Repeating Rectangle Layout: RapidLayout iteratively partitions one SLR and calculates utilization until the divided section does not fit any unit. Then, the partition scheme with the highest utilization rate is selected to determine the repeating rectangular region. In Fig. 6, we show the floorplan for such a region. The resource utilization within the rectangle is 100% URAMs, 93.7% DSP48s, and 95.2% BRAMs, which holds for the entire chip after replication. Our optimizer minimizes overlapping and thus reducing routing congestions to permit high-frequency operation.

Full-Chip Layout: In Fig. 7, we show the full-chip layout for the systolic array accelerator. The entire chip layout is generated in three steps: (1) First, the rectangular region's placement is replicated to fill up one SLR (SLR0). The SLR is then pipelined and fully routed. (2) Second, the placement and routing from SLR0's implementation

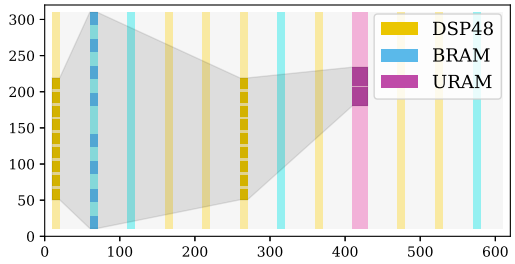


Fig. 5. Floorplan layout visualization of a single convolution block implementation supporting dual 3×3 kernels to match URAM bandwidth. This is the design shown in Fig. 2 earlier. The bounding polygon that encloses all hard blocks and the routing connections is shown in gray.

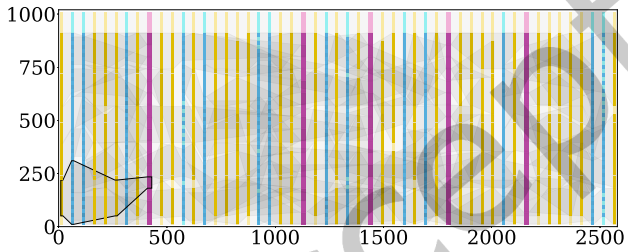


Fig. 6. Floorplan layout visualization of a single repeating rectangular region layout with 80 convolution blocks. The bounding polygon from Fig. 5 is also shown here for scale.

are replicated across the two other SLRs to fill up the FPGA. (3) Finally, RapidLayout generates SLR bridges for cross-SLR connections.

3.4 SLR Crossing

The connections between systolic array building blocks create cross-SLR routes during replication. SLR crossing faces two challenges: (1) the variability between silicon dies may cause high clock skew for transmitting and receiving registers, causing hold or setup time violations. (2) Cross-SLR datapaths have additional inter-SLR compensation latency [12]. The increased path delay makes cross-SLR paths more critical to overall design timing performance.

Xilinx UltraScale+ Devices [51] provide dedicated Laguna Sites for cross-SLR routing. We reuse RapidWright's `SLRCrosserGenerator` [26] to create Laguna TX/RX Flip-flop pairs and perform custom clock routing. RapidLayout generates SLR bridges in three steps:

- **Select Laguna Sites**: identify hard block pairs with cross-SLR connections, and select the nearest Laguna Site pairs to create SLR bridges.
- **Create Super-Long Lines (SLL)**: create, place, and route Laguna Flip-flops, then custom route clock signals with `SLRCrosserGenerator` utilities.
- **Route SLR Bridges**: route hard block pairs with Laguna flip-flops with a depth-first search for routing channel between source and sink nodes.

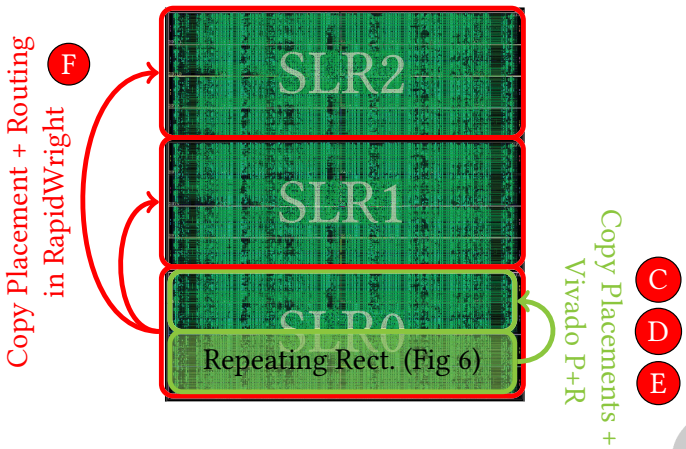


Fig. 7. Full-chip layout for the systolic array accelerator generated from a repeating rectangle of size two clock regions high and full-chip wide. After one replication we span one SLR region. We place and route this with Vivado, export DCP, reimport into RapidWright to clone across SLRs and generate cross-SLR connections. Step C: hard block placement and site routing. Step D: post-placement pipelining. Step E: SLR Placement and routing in Vivado. Step F: SLR Replication.

To overcome high clock skew, we drive Laguna TX/RX Flip-flop's clock signal from leaf clock buffers on both sides of the SLR bridge. Specifically, we create a global `BUFGCE` clock buffer and route output clock signals to `NODE_GLOBAL_LEAF` nodes in boundary clock regions.

3.4.1 Optimize for Cross-SLR Clock Frequency. Cross-SLR connections are critical for the final achievable clock frequency. When creating SLR bridges, RapidLayout chooses the nearest Laguna sites to minimize routing path delay. Additionally, we can add an objective to minimize cross-SLR wirelength during placement optimization:

$$\min \sum_{i \in H} (|x_i - LAGUNA\_X_i| + |y_i - LAGUNA\_Y_i|) \tag{6}$$

In the objective expression above:

- $H \subset \{DSP, BRAM, URAM\}$ denotes hard blocks with cross-SLR connections.
- $LAGUNA\_X_i$ and $LAGUNA\_Y_i$ denote the RPM coordinates of the nearest Laguna site to hard block i.

Similar to Equation 1, we minimize the Manhattan distance between hard blocks and the nearest Laguna sites. The difference is that the nearest Laguna site is chosen based on hard block i's location, instead of decoded from the genotype.

3.5 Placement Transfer Learning

RapidLayout is capable of delivering high-quality placement results on devices with different sizes, resource ratios, or column arrangements with transfer learning ability. Transfer learning uses the genotype of an existing placement as a starting seed for placement search on a new device. For devices with significant resources difference, we also explore the opportunity of transfer learning with placement bootstrap.

3.5.1 Placement Transfer Learning from Similar Devices. Transfer learning applies to devices with similar amounts of hard block resources. Specifically, source and target devices should have enough hard blocks to fit the same number of computation units. Transfer learning between devices with similar capacities is straightforward: (1) encode source placement into genotype, (2) decode genotype according to the target device. Since RapidLayout's

evolutionary genotype encodes placement with distribution and relative location of blocks in their column, solutions can easily transfer to new devices with similar hardware resources but different column arrangements.

Algorithm 1: Mapping genotype extrapolation

```
Input  : Source Mapping Genotype: input_mapping
Input  : Target length: target_len
Output : Target Mapping Genotype: output_mapping
source_len = len(input_mapping);
n = Floor(target_len / source_len) + 1;
for i ← 0 to n do
    offset = i × source_len;
    foreach Integer in input_mapping do
        if offset + Integer < target_len - 1 then
            output_mapping.add(offset + Integer);
```

3.5.2 Placement Bootstrap from Smaller Devices. Placement bootstrap is to obtain high-quality initialization on a large target device from searched solutions on smaller source devices. We propose a genotype extrapolation method for placement bootstrap. From the discussion in Section 4.2.2, we show that optimizing mapping genotype alone can still achieve good performance. When we only optimize mapping, the hard-block column distribution is generated as uniform, and hard blocks are selected from bottom to top within each column. Similarly, to obtain genotype encoding from smaller placement, we can extrapolate the mapping genotype alone, and set distribution and location genotype as uniform and stacking-from-bottom. The mapping genotype extrapolation method is detailed in Algorithm 1.

Understanding mapping genotype extrapolation: The mapping genotype is composed of three integer lists whose orders are optimized, each corresponds to DSP, BRAM, and URAM blocks. The order of the three lists determines the connectivity of the chosen physical blocks. A placement solution with small wirelength and bounding box sizes would limit the hard block connections to as local as possible, forming meaningful patterns in the three lists. Therefore, the extrapolation algorithm repeats the sequence of the source mapping genotype to preserve the local order patterns.

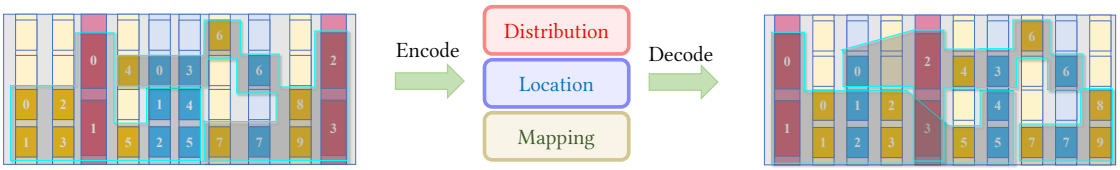
3.6 Transfer Learning Example Walkthrough

We provide examples of transfer learning to demonstrate how existing placement is transferred from devices with similar or smaller capacities. We walk you through two scenarios: (1) source and target devices have a similar number of hard blocks but different column arrangements, (2) both devices are of different sizes and also have different column arrangements.

3.6.1 Transfer learning from devices with similar capacities. Fig. 8 (a) and (c) show the source and target placement of two computation units, each consists of 5 DSPs, 4 BRAMs, and 2 URAMs. The source and target devices have the same number of hard block columns but different arrangements.

Placement encoding: We construct distribution and location genotypes from the number and relative position of blocks in their columns. Next, we concatenate the numeric label of hard blocks used by each computation unit and obtain the mapping genotype.

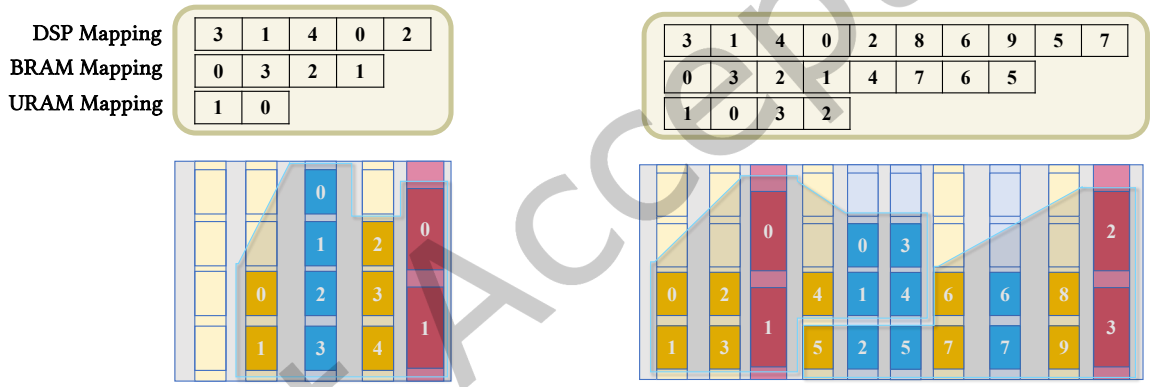
Placement decoding: After that, we encode the source placement into genotypes, we decode the genotypes with respect to the target device. First, we quantize distribution and location genotypes to select hard blocks in



(a) Source placement (b) Genotype (c) Target placement

Fig. 8. Transfer learning example: source and target devices have a similar amount of resources. (a) Source placement is encoded to (b) genotype, then decoded on (c) target device.

each column. Then, we number the selected blocks and connect them according to the mapping genotype. We can easily transfer to devices with different resource arrangements because of our genotype formulation.



(a) Source placement (1 computation unit) (b) Target placement (2 computation units)

Fig. 9. Placement bootstrap from a smaller device with different resource distribution. Placement bootstrap allows source and target devices to have drastically different capacity and resource distribution. (a) Small placement on source device with 5 DSPs, 4 BRAMs, and 2 URAMs. Shadow shows one unit's placement and enclosing routing resources. (b) Extrapolated placement initialization on the target device with two such computation units. Extrapolated placement serves as a high-quality initialization on the target device to reduce runtime and obtain better results.

3.6.2 Bootstrapping from smaller devices. We show an example of transferring placement from one computation unit to a larger device that fits two units in Fig. 9. The target device is about 2× larger than the source device, with different column arrangements.

Genotype extrapolation: The extra step for placement bootstrap is to extrapolate the genotype to represent a larger placement. During genotype extrapolation, distribution and location genotypes are generated according to target device size, thus omitted from Fig. 9. For mapping genotypes, we replicate the source mapping genotypes, add an offset of its length, and concatenate the result with the source mapping genotype. For a target placement N times larger than source placement, we perform this operation for $N - 1$ times.

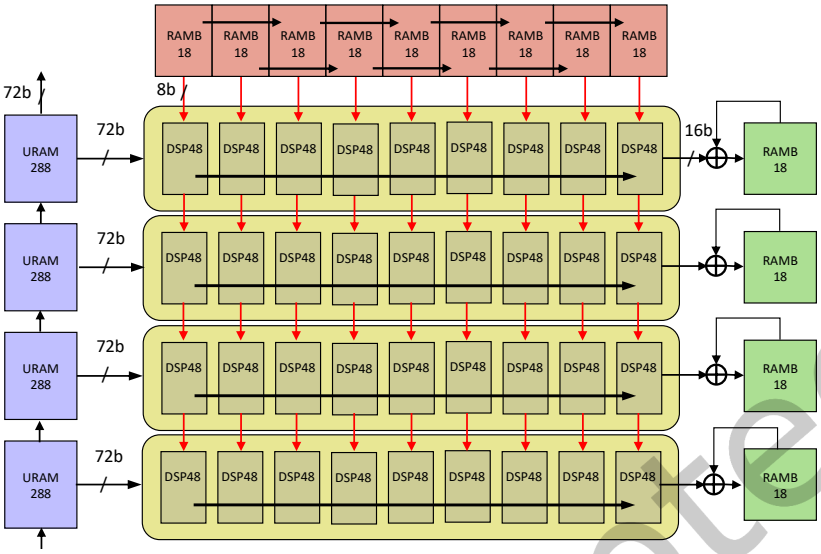


Fig. 10. The matrix-vector multiplication tile proposed in [40] as a case study for RapidLayout's generality. Each tile is a computation unit during placement. The multiplication tile consists of one URAM chain for matrix streaming, one BRAM chain for vector streaming, four DSP chains for MAC, and four free BRAMs for result collection.

3.7 Generality of RapidLayout

RapidLayout's evolutionary placement method is not targeting a particular design, but a range of tiled hard-block modular designs. As discussed in Section 3.1.2, RapidLayout's evolutionary genotype encoding applies to general hard block designs. For tiled designs with a large number of computing units, RapidLayout can exploit the regularity to reduce problem size and accelerate placement optimization. If the FPGA device has multiple SLRs, RapidLayout could further accelerate implementation through reusing routing in a single SLR.

3.7.1 Case Study: Matrix-Vector Multiplication Systolic Array. We take the matrix-vector multiplication systolic array from [40] as an example to show RapidLayout's capability to handle general hard block designs. As shown in Fig. 10, the computing unit has 4 cascaded URAMs, 9 cascaded BRAMs, another 4 BRAMs that are not cascaded, and 4 matrix-vector blocks each with 9 cascaded DSPs. RapidLayout is able to deliver high-quality placement for such tiled design with both cascaded and free hard blocks.

For any modular hard block design unit C_k, the following inputs are necessary to calculate the objective functions defined as Equation 1 and Equation 2: (1) The hard block instances in the cascade chain, and the length of the cascade chain. Hard blocks that are not cascaded are treated as a length-1 cascade chain. (2) The connections between hard blocks, including source and sink block names and the connection width. (3) The target device. With these inputs, we run RapidLayout workflow for the matrix-vector multiplication systolic array. As shown in Fig. 11, RapidLayout identifies a repeating rectangle that fits 40 blocks to search for a placement. The placement optimization with NSGA-II algorithm costs 594 seconds, and obtains a solution with $wirelength = 3288.3$ and $maxBBoxSize = 1899$. The final routed design's clock frequency is 668 MHz on Xilinx VU11P device, and the entire workflow costs 68 minutes. Compared with the convolutional systolic array design presented in Section 2.1, the matrix multiplication tile in this case study uses 2× more URAM and DSP, 1.625× more BRAM with different cascade configurations. RapidLayout still delivers a routed design with above 650 MHz clock frequency in ≈1 hour.

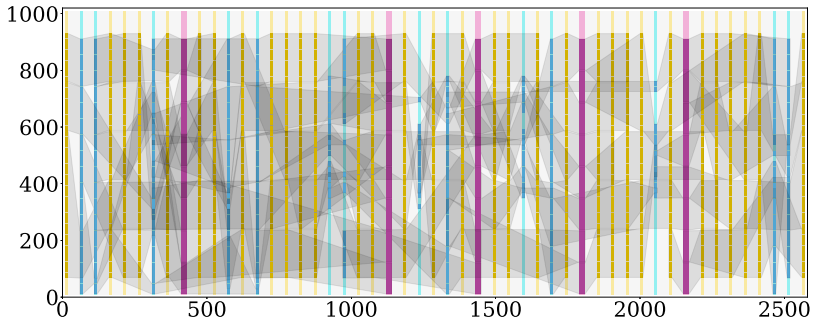


Fig. 11. Searched placement for 40 matrix multiplication blocks with NSGA-II on Xilinx UltraScale+ VU11P. The final routed design achieves 668 MHz clock frequency with a complete workflow runtime of 63 minutes.

3.8 Target Scope of RapidLayout

With the case study presented in Section 3.7.1, we show that RapidLayout is capable of producing high-quality placement for designs with different hard block ratios, cascade configurations, and connectivity. In this section, we discuss the target FPGA architectures and circuit designs to define the tool's scope, and we discuss RapidLayout's scalability to other FPGA architectures and designs.

3.8.1 Target FPGA Architectures. The implementations in this work target the Xilinx UltraScale+ FPGA architecture. However, RapidLayout's problem formulation and evolutionary placement algorithm can be applied to other FPGA architectures from different vendors as well. We take Intel's traditional Agilex architecture [6] and the latest AI-optimized Stratix 10 NX architecture [4] as examples, and we also discuss RapidLayout's applicability on Xilinx's Versal architecture [13].

FPGA architectures with four traits could benefit from RapidLayout:

(1) Hard blocks are arranged in columns. RapidLayout's genotype encoding assumes vertically arranged hard blocks, as discussed in Section 3.1.
(2) Different types of hard block columns are spread unevenly across the device. This situation constraints us from horizontally copy-pasting a unit placement. Therefore, RapidLayout interleaves units to get a good placement across irregular hard block columns.
(3) Hard blocks in the same column can be cascaded for fast data movement. This characteristic of architecture is optional. But if an architecture supports cascades, RapidLayout can handle the cascade constraints and exploit this feature for higher clock frequencies.
(4) Vertical regularity. This means the columnar arrangement of hard blocks repeats vertically. This is also an optional feature, but it allows RapidLayout to identify a minimal repeating region to reduce the problem size, and to reuse placement and routing results.

For example, Intel's Agelix FPGAs [6] have DSPs and RAMs arranged in columns and its DSPs support cascades in INT9 mode. Their configurable fabrics are divided into sectors, which provides regularity to reduce placement problem size and reuse results. Intel's Stratix 10 NX [4] with AI tensor blocks can also benefit from RapidLayout. The AI tensor blocks on the device are also arranged in columns and support cascades in INT8 mode. The Xilinx Versal architecture [13] offers a higher level of regularity by repeating the clock regions horizontally at regular intervals. With this feature, RapidLayout could further minimize the repeating rectangular region and gain more implementation speedups. However, the specialized AI Engines for neural network workloads are separated from

the configurable fabric in the floorplan. Since the hardened AI Engines are not mixed with other hard blocks, the floorplanning for them is beyond the scope of RapidLayout. In general, RapidLayout is suitable for devices with heterogeneous hard blocks, where the uneven distribution and density of the hard blocks pose difficulty to high-utilization design placements.

3.8.2 Target Circuit Designs. RapidLayout is applicable for circuits with many identical computation units, or Processing Elements (PE), where each PE uses a set of different hard blocks and has its own control logic. As discussed in Section 3.7.1, RapidLayout takes the design's logic netlist as input, along with the information of hard block connections and cascade configurations. The other assumption is that the target design does not contain complex global control logic. Systolic arrays are such examples, where PEs are identical and have local control circuits. RapidLayout leaves soft logic placement to Vivado and only optimizes the placement of hard blocks in the computation units. If the design contains large and complex global control logic, the critical path is likely to appear in the control logic where RapidLayout cannot help.

4 RESULTS

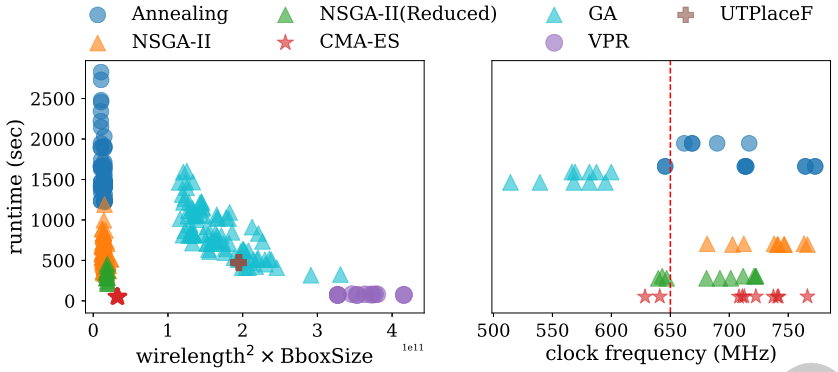
RapidLayout is implemented in Java to enable seamless integration with RapidWright Java APIs. We use the Java library Opt4J [30] as the optimization framework for NSGA-II, SA, and GA. CMA-ES is implemented with Apache Commons Math Optimization Library [34] 3.4 API. We use VPR 7.0 official release [31] and UTPlaceF TCAD version [28] binary for QoR comparison. All placed designs are routed and timed with Vivado Design Suite 2018.3. We run our experiments on an Ubuntu 16.04 LTS machine with Intel Xeon Gold 5115 CPU (10 cores, 20 threads) and 128 GB DDR4 RAM.

4.1 Performance and QoR Comparison

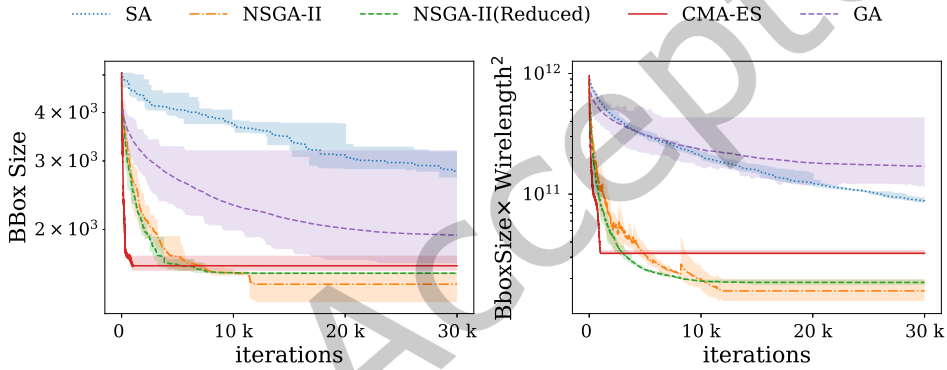
We compare the performance and QoR of evolutionary algorithms against (1) conventional simulated annealing (SA), (2) academic placement tool VPR 7.0, (3) state-of-art analytical placer UTPlaceF, (4) single-objective genetic algorithm (GA) [54], and (5) manual placement design. We exclude RapidWright's default annealing-based block placer since it does not give feasible placement solutions. We run each placement algorithm 50 times with seeded random initialization. Then, we select the top-10 results for each method to route and report clock frequency. While we include VPR and UTPlaceF in comparison, they do not support **cascade** constraints (Equation 5). This limits our comparison to an approximate check on solution quality and runtime, and we are unable to translate the resulting placement to the FPGA directly.

In Fig. 12a, we plot total runtime and final optimized wirelength^2 $\times$ maximum bounding box size for the different placement algorithms along with Vivado-reported frequency results. We see some clear trends: (1) NSGA-II is ≈2.7× faster than SA and delivers 1.2× bounding box improvement, but has ≈12.9% longer wirelength. The average clock frequency of top-10 results is evidently higher than SA as NSGA-II's performance is more stable. (2) CMA-ES is ≈30× faster than SA. Although the average bounding box size (≈16% larger) and wirelength (≈42% larger) are worse than SA's results, CMA-ES achieves a slightly higher average clock frequency at 711 MHz. (4) An alternate NSGA-II method discussed later in Section 4.2.2 with a reduced search space delivers roughly 5 times shorter runtime than SA, with only 2.8% clock frequency degradation, which is still above the URAM-limited 650 MHz maximum operating frequency.

In Fig. 12b, we see the convergence rate of the different algorithms when observing bounding box sizes and the combined objective. NSGA-II clearly delivers better QoR after 10 k iterations, while CMA-ES delivers smaller bounding box sizes within a thousand iterations. Across multiple runs, bounding box optimization shows a much more noisy behavior with the exception of CMA-ES. This makes it (1) tricky to rely solely on bounding box minimization, and (2) suggests a preference for CMA-ES for containing critical paths within bounding boxes.



(a) Wirelength, Bounding Box vs. Runtime Comparison



(b) Effect of Convergence on Wirelength, Bounding Box

Fig. 12. Performance, Wirelength, and Bounding Box Comparison: Tuned Simulated Annealing (SA, Annealing), NSGA-II, NSGA-II (Reduced), CMA-ES, Genetic Algorithm (GA), VPR, and UTPlaceF. NSGA-II (Reduced) is further discussed in Section 4.2.2.

Table 1

Runtime(avg), Wirelength(avg), Max BBox(avg), Pipelining Registers(min), and Frequency(avg) for all methods. NSGA-II shows reduced genotype as well. Speedups and QoR improvements wins by Evolutionary algorithms are also reported in red→NSGA-II and green→CMA-ES for each competitor algorithm (SA, GA, UTPlaceF, VPR, Manual).

Method	NSGA-II	CMA-ES	SA	GA	VPR	UTPlaceF	Manual
Runtime (secs)	586 (323)	51	1577 (2.7×, 30.8×)	850 (1.5×, 16.7×)	76 (0.13×, 1.5×)	473 (0.8×, 9.3×)	1–2 wks
Wirelength	3.5K (3.5K)	4.4K	3.1K (0.9×, 0.7×)	9.2K (2.6×, 2.1×)	8.5K (2.4×, 1.9×)	7.8K (2.2×, 1.8×)	8.1K (2.3×, 1.8×)
BBox	1183 (1543)	1606	1387 (1.2×, 0.9×)	1908 (1.6×, 1.2×)	4941 (4.1×, 3.1×)	3218 (2.7×, 2.0×)	1785 (1.5×, 1.1×)
Pipeline Reg.	256K (273K)	273K	273K (1.1×, 1×)	323K (1.3×, 1.2×)	-	-	306K (1.2×, 1.1×)
Frequency (MHz)	733 (688)	708	711 (1.03×, 0.99×)	585 (1.3×, 1.2×)	-	-	693 (1.1×, 1.02×)

Finally, in Table 1, we compare average metric values across the 50 runs of all methods. NSGA-II and CMA-ES deliver superior QoR and runtime against UTPlaceF and VPR, and speed up runtime by 3–30× against annealing with a minor loss in QoR. Table 1 also reports the number of registers needed for the 650 MHz operations. NSGA-II delivers this with 17k (≈6%) fewer registers against annealing and 50k (≈16%) fewer registers against manual placement. NSGA-II results in Table 1 are run in 20 threads. Although CMA-ES runs in serial, the runtime is ≈10× faster than NSGA-II with a QoR gap.

4.2 Parameter Tuning for Annealing and NSGA-II

In this section, we discuss cooling schedule selection for annealing and optimizations to NSGA-II to explore quality, runtime trade-offs.

4.2.1 Parameter Tuning for SA. The cooling schedule determines the final placement quality, but it is highly problem-specific. We plot the cooling schedule tuning process in Fig. 13 and choose the hyperbolic cooling schedule for placement experiments presented in Table 1 to achieve the best result quality.

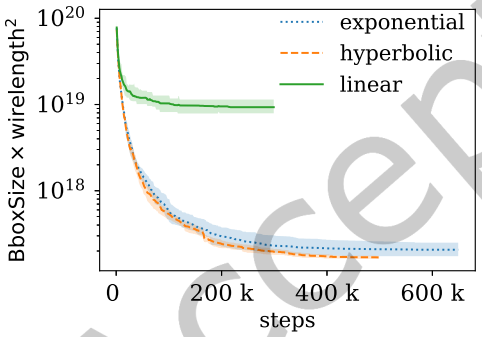


Fig. 13. SA Parameter Tuning. Each cooling schedule is run with 10 sets of parameters. Annealing placement experiments presented in Table 1 use a hyperbolic cooling schedule for the best QoR performance.

4.2.2 NSGA-II Reduced Genotype. As per the genotype design, distribution and location genotypes take up a large portion of the composite genotype, and they demand quantization and legalization steps. However, for high-utilization designs, distribution and location are less influential since resources are nearly fully utilized. Therefore, we reduce the genotype to mapping only for NSGA-II, and uniformly distribute and stack the hard blocks from bottom to top. As a consequence of this trade-off, we observe a ≈1.8× runtime improvement but a 1.3× larger bounding box size against the original NSGA-II. In the convergence plot of Fig. 12b, we discover that reduced genotype does not save iteration needed, and the bulk of the runtime improvements comes from reduced genotype decoding and legalization work.

4.3 Pipelining

Finally, we explore the effect of pipelining stages on different placement algorithms. At each pipelining depth, multiple placements from each algorithm are routed by Vivado to obtain a frequency performance range.

In Fig. 14, we show the improvement in frequency as a function of the number of pipeline stages inserted along the long wires by RapidLayout. We note that NSGA-II delivers 650 MHz frequency with no pipelining, while others require at least one stage. Therefore, NSGA-II saves ≈6%–16% registers at pipelining as shown in Table 1. NSGA-II wins over manual design at every depth, and CMA-ES exhibits the most stable performance.

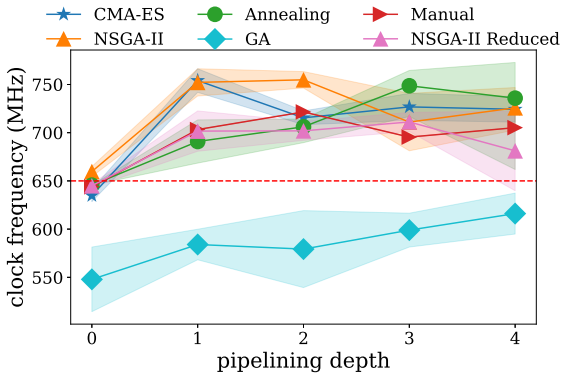


Fig. 14. Effect of post-placement pipelining on the clock frequency of the design. NSGA-II delivers 650 MHz without extra pipelining, while CMA-ES, Annealing, and Manual placement require at least one stage. NSGA-II and CMA-ES achieve 750+ MHz with two stages, while SA requires four stages.

Systolic array operation at 750+ MHz should be possible with planned future design refinements. CMA-ES and NSGA-II can deliver 750+ MHz frequency with only two pipeline stages, while SA requires four stages.

4.4 SLR Bridge

Table 2
SLR Crosser Performance: max clock frequency, clock skew, and Inter-SLR Compensation (ISC).

Device	Size	Clk Freq. (MHz)	Clk Skew (ns)	ISC (ns)
xcvu9p	181-bit×10	909.1	-0.192	0.138
xcvu11p	181-bit×10	911.6	-0.193	0.135
xcvu13p	181-bit×15	911.6	-0.193	0.135

We generate SLR bridges for cross-SLR URAM cascade signals. In Table 2, we evaluate the generated SLR bridges in terms of maximum clock frequency, clock skew, and Inter-SLR Compensation (ISC) across multi-die UltraScale+ devices. We observe that the TX/RX Laguna FF pairs support beyond 900 MHz clock frequencies. Driving Laguna FF's clock signals with `NODE_GLOBAL_LEAF` leaf clock buffers enables low clock skew and low ISC penalty within ±0.2ns.

4.5 Transfer Learning

We partition Xilinx UltraScale+ family into two groups with a similar number of hard block columns. We choose VU3P and VU11P as "seed" devices on which RapidLayout generates placement from scratch with NSGA-II. Thereafter, placement results on seed devices are migrated to destination devices in the same group. In Table 3 and Fig. 15, we compare placement runtimes with and without transfer learning across a range of FPGA device sizes. We observe that transfer learning accelerates the optimization process by 11–14× with a frequency variation from -2% to +7%. If we observe the total implementation runtime column, we note that SLR replication ensures that the increase in overall runtime (46 mins.→69 mins., 1.5×) with device sizes much smaller than the FPGA capacity increase (123→640, 5.2×).

Table 3
Transfer Learning Performance: VU3P, VU11P as Seed Devices

Device	Design Size (conv units)	Impl.Runtime (mins.)	Frequency (MHz)		Placement Runtime (s)	
			Scratch	Transfer	Scratch	Transfer
xcvu3p	123	46.4	718.9	-	428.3	-
xcvu5p	246	56.9	677.9	660.5	577.9	42.2 (13.7×)
xcvu7p	246	55.1	670.2	690.1	578.8	41.9 (13.8×)
xcvu9p	369	58.4	684.9	662.3	570.8	42.0 (13.6×)
xcvu11p	480	65.2	655.3	-	522.4	-
xcvu13p	640	69.4	653.2	701.3	443.7	38.4 (11.6×)

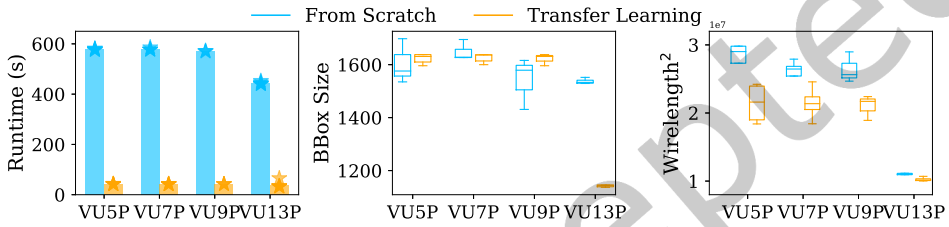


Fig. 15. Runtime and QoR comparison between running from scratch and transfer learning. Transfer learning delivers ≈11–14× faster runtime, 0.95–1.3× bbox size improvement, and 1.05–1.17× wirelength improvement

4.6 Placement Bootstrap

We use a device with 5 × 24 DSP48E2, 3 × 24 RAMB18, and 1 × 16 URAM288 blocks as a "seed" device to generate source placement. Then, we extrapolate the source placement to VU3P-VU13P as bootstrap initialization. In Table 4 and Fig. 16, we compare the QoR and runtime of placement bootstrap and random initialization across a range of target devices. We observe that bootstrap shortens placement optimization runtime by 2.1–3.2×, and improves bounding box size by 1.2–1.6×, and wirelength by 0.95–1.6×. The improvement in QoR leads to higher clock frequency for routed designs (0.98–1.12×).

Table 4
Placement Boostrap Performance: bootstrap from 5-block placement vs random initialization

Device	Target Pl. Size (conv units)	Impl.Runtime (mins.)	Frequency (MHz)		Placement Runtime (s)	
			Random Init.	Bootstrap	Random Init.	Bootstrap
xcvu3p	123	46.9	718.9	726.2 (1.01×)	428.3	158.1 (2.7×)
xcvu5p	123	56.8	677.9	670.2 (0.98×)	577.9	245.7 (2.4×)
xcvu7p	123	57.2	670.2	745.2 (1.11×)	578.8	265.3 (2.2×)
xcvu9p	123	58.5	684.9	745.2 (1.09×)	570.8	273.8 (2.1×)
xcvu11p	160	65.9	655.3	735.8 (1.12×)	522.4	166.2 (3.1×)
xcvu13p	160	68.6	653.2	719.9 (1.10×)	443.7	138.2 (3.2×)

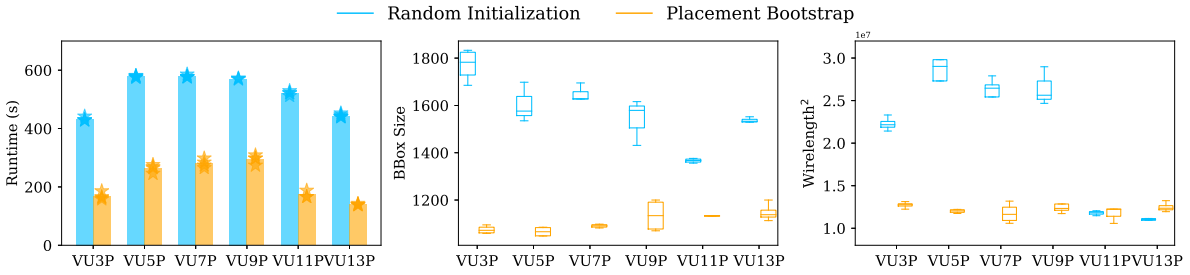


Fig. 16. Runtime and QoR comparison between placement bootstrap and random initialization. Placement bootstrap delivers 2.1-3.2× faster runtime, 1.2-1.6× bounding box size improvement, and 0.95-1.6× wirelength improvement. Placement bootstrap also improved clock frequency by 0.98-1.12×.

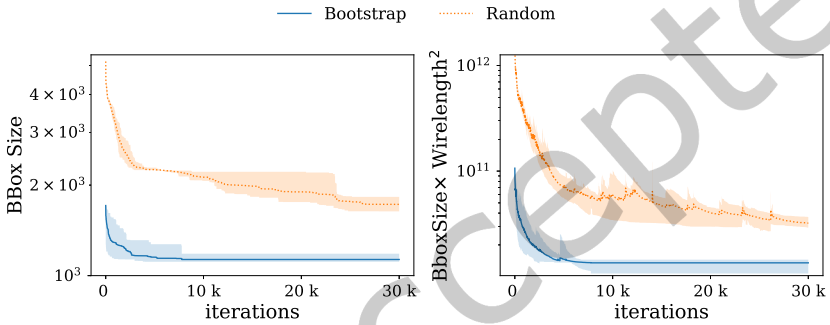
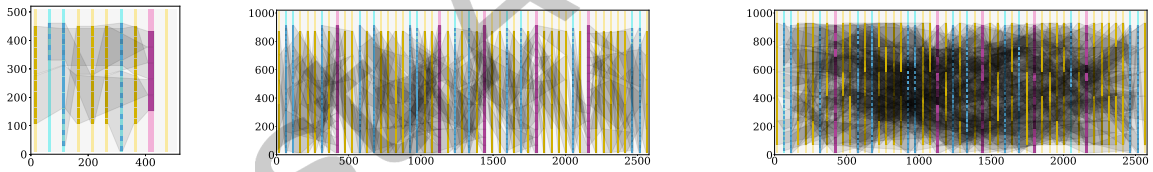


Fig. 17. Wirelength and bounding box size comparision between placement bootstrap and random initialization.



(a) Source (b) Placement bootstrap (c) Random initialization

Fig. 18. Placement bootstrap as initialization versus random initialization. (a) The source placement that only fits 5 convolution blocks. (b) Placement bootstrap from a small source device to UltraScale+ VU11P (80 convolution blocks). (c) Random initialization on the target device (80 convolution blocks).

We visualize the placement bootstrap initialization in Fig. 18 to understand its advantage against random initialization. We show that repeating the local pattern of a smaller placement is an effective method to generate high-quality initial placement on larger devices. Compared with random initialization (Fig. 18(c)), bootstrapping from small placement (Fig. 18(b)) offers much lower congestion and smaller bounding box sizes. We compare the wirelength and bounding box size over search iterations for optimizing from bootstrap and random initializations in Fig. 17. We observe that searching from bootstrap offers a ≈3× smaller initial bounding box size, which contributes to shorter optimization runtime and better QoR.

5 CONCLUSIONS

We present an end-to-end hard block placement workflow for resource-intensive systolic array designs on modern heterogeneous FPGAs. RapidLayout delivers an automatic placement-and-routing workflow ≈5–6× faster than Vivado that eschews manual placement effort. Evolutionary algorithms outperform Simulated Annealing, VPR's annealer, UTPlaceF analytical placer, and Manual Placement by 1.5–30.8× in runtime, 1.8–2.4× in wirelength and 1.1–4.1× in bounding box sizes. RapidLayout also enables transfer learning from devices with similar capacity and placement bootstrap from much smaller devices. Transfer learning achieves 11–14× faster optimization, and bootstrapping from small devices delivers 2.1–3.2× shorter runtime. RapidLayout code is available at https://git.uwaterloo.ca/watcag/rapidlayout.

REFERENCES

[1] Ziad Abuowaimer, Dani Maarouf, Timothy Martin, Jeremy Foxcroft, Gary Gréwal, Shawki Areibi, and Anthony Vannelli. 2018. GPlace3. 0: Routability-driven analytic placer for UltraScale FPGA architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23, 5 (2018), 1–33.

[2] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications*. Springer, 213–222.

[3] V Betz, J Rose, and A Marquardt. 1999. Architecture and CAD for Deep-Submicron FPGAs, chapter Appendix B.

[4] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C Hoe, Vaughn Betz, and Martin Langhammer. 2020. Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 10–19.

[5] Xilun Chen, Ahmed Hassan Awadallah, Hany Hassan, Wei Wang, and Claire Cardie. 2018. Multi-source cross-lingual model transfer: Learning what to share. *arXiv preprint arXiv:1810.03552* (2018).

[6] Jeffrey Chromczak, Mark Wheeler, Charles Chiasson, Dana How, Martin Langhammer, Tim Vanderhoek, Grace Zgheib, and Ilya Ganusov. 2020. Architectural enhancements in intel® agilexâĎć fpgas. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 140–149.

[7] Robert Collier, Christian Fobel, Laura Richards, and Gary Grewal. 2012. A formal and empirical analysis of recombination for genetic algorithm-based approaches to the FPGA placement problem. In *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 1–6.

[8] Felipe Leno Da Silva and Anna Helena Reali Costa. 2019. A survey on transfer learning for multiagent reinforcement learning systems. *Journal of Artificial Intelligence Research* 64 (2019), 645–703.

[9] Jeffrey Dean. 2019. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design. *arXiv preprint arXiv:1911.05289* (2019).

[10] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[11] K. Eguro and S. Hauck. 2008. Simultaneous Retiming and Placement for Pipelined Netlists. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. 139–148. https://doi.org/10.1109/FCCM.2008.21

[12] Tom Feist. [n. d.]. *Xilinx White Paper: Vivado Design Suite (WP416)*. Xilinx.

[13] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: VersalTM architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 84–93.

[14] Kasthurirangan Gopalakrishnan, Siddhartha K Khaitan, Alok Choudhary, and Ankit Agrawal. 2017. Deep convolutional neural networks with transfer learning for computer vision-based data-driven pavement distress detection. *Construction and building materials* 157 (2017), 322–330.

[15] Marcel Gort and Jason H Anderson. 2012. Analytical placement for heterogeneous FPGAs. In *22nd international conference on field programmable logic and applications (FPL)*. IEEE, 143–150.

[16] Nikolaus Hansen and Andreas Ostermeier. 1996. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of IEEE international conference on evolutionary computation*. IEEE, 312–317.

[17] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*. PMLR, 2790–2799.

[18] Peter Jamieson. 2010. Revisiting Genetic Algorithms for the FPGA Placement Problem.. In *GEM*. 16–22.

[19] Peter Jamieson. 2011. Exploring inevitable convergence for a genetic algorithm persistent fpga placer. In *Proceedings of the International Conference on Genetic and Evolutionary Methods (GEM)*. The Steering Committee of The World Congress in Computer Science, Computer âĂę, 1.

[20] Peter Jamieson, Farnaz Gharibian, and Lesley Shannon. 2013. Supergenes in a genetic algorithm for heterogeneous FPGA placement. In *2013 IEEE Congress on Evolutionary Computation*. IEEE, 253–260.
[21] Jing Jiang. 2009. Multi-task transfer learning for weakly-supervised relation extraction. ACL.
[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12. https://doi.org/10.1145/3079856.3080246
[23] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.
[24] Simon Kornblith, Jonathon Shlens, and Quoc V Le. 2019. Do better imagenet models transfer better?. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2661–2671.
[25] Kung. 1982. Why systolic architectures? *Computer* 15, 1 (Jan 1982), 37–46. https://doi.org/10.1109/MC.1982.1653825
[26] Chris Lavin and Alireza Kaviani. 2018. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 133–140.
[27] Kaiwen Li, Tao Zhang, and Rui Wang. 2019. Deep Reinforcement Learning for Multi-objective Optimization. *arXiv preprint arXiv:1906.02386* (2019).
[28] Wuxi Li, Shounak Dhar, and David Z Pan. 2017. UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 4 (2017), 869–882. http://wuxili.net/project/utplacef/
[29] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. 2019. Nsga-net: neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 419–427.
[30] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. 2011. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)* (July 12–16). Dublin, Ireland, 1723–1730.
[31] Jason Luu, Jeff Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Norrudin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 7, 2 (June 2014), 6:1–6:30. https://verilogtorouting.org/download/
[32] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. 2003. Fast timing-driven partitioning-based placement for island style FPGAs. In *Proceedings of the 40th annual design automation conference*. 598–603.
[33] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. 2005. Timing-driven partitioning-based placement for island style FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 3 (2005), 395–406.
[34] Apache Commons Math. 2013. Commons Math: The Apache Commons Mathematics Library.
[35] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191
[36] Ariadna Quattoni, Michael Collins, and Trevor Darrell. 2008. Transfer learning for image classification with sparse prototype representations. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1–8.
[37] Raymond Ros and Nikolaus Hansen. 2008. A simple modification in CMA-ES achieving linear time and space complexity. In *International Conference on Parallel Problem Solving from Nature*. Springer, 296–305.
[38] Sebastian Ruder, Matthew E Peters, Swabha Swayamdipta, and Thomas Wolf. 2019. Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*. 15–18.
[39] Krzysztof Rusek, JosÃľ SuÃąrez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2020. RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2260–2270. https://doi.org/10.1109/JSAC.2020.3000405
[40] A. Samajdar, T. Garg, T. Krishna, and N. Kapre. 2019. Scaling the Cascades: Interconnect-aware FPGA implementation of Machine Learning problems. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.
[41] Kun Shao, Yuanheng Zhu, and Dongbin Zhao. 2018. Starcraft micromanagement with reinforcement learning and curriculum transfer learning. *IEEE Transactions on Emerging Topics in Computational Intelligence* 3, 1 (2018), 73–84.
[42] Y. Shen, M. Ferdman, and P. Milder. 2016. Overcoming resource underutilization in spatial CNN accelerators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–4. https://doi.org/10.1109/FPL.2016.7577315
[43] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 535–547. https://doi.org/10.1145/3079856.3080221

[44] Deshanand P. Singh and Stephen D. Brown. 2002. Integrated Retiming and Placement for Field Programmable Gate Arrays. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA âĂŹ02)*. Association for Computing Machinery, New York, NY, USA, 67âĂŞ76. https://doi.org/10.1145/503048.503059

[45] Matthew E Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10, 7 (2009).

[46] Steven Trimberger and M-R Chene. 1992. Placement-based partitioning for lookup-table-based FPGAs. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*. IEEE, 91–94.

[47] Rahman Venkatraman and Lalit M Patnaik. 2000. An evolutionary approach to timing driven fpga placement. In *Proceedings of the 10th Great Lakes symposium on VLSI*. 81–85.

[48] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. 2003. Post-Placement C-Slow Retiming for the Xilinx Virtex FPGA. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA âĂŹ03)*. Association for Computing Machinery, New York, NY, USA, 185âĂŞ194. https://doi.org/10.1145/611817.611845

[49] E. Wu, X. Zhang, D. Berman, and I. Cho. 2017. A high-throughput reconfigurable processing array for neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–4. https://doi.org/10.23919/FPL.2017.8056794

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

[51] Xilinx. [n. d.]. *Large FPGA Methodology Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/ug872_largefpga.pdf

[52] Xilinx. [n. d.]. *Vivado Design Suite User Guide, Using Constraints (UG903)*. Xilinx.

[53] Junjie Xing, Kenny Zhu, and Shaodian Zhang. 2018. Adaptive multi-task transfer learning for Chinese word segmentation in medical text. In *Proceedings of the 27th International Conference on Computational Linguistics*. 3619–3630.

[54] M Yang, AEA Almaini, L Wang, and PJ Wang. 2005. An evolutionary approach for symmetrical field programmable gate array placement. In *Research in Microelectronics and Electronics, 2005 PhD*, Vol. 1. IEEE, 169–172.

[55] Cunxi Yu and Zhiru Zhang. 2019. Painting on placement: Forecasting routing congestion using conditional generative adversarial nets. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[56] Xin Zheng, Luyue Lin, Bo Liu, Yanshan Xiao, and Xiaoming Xiong. 2020. A multi-task transfer learning method with dictionary learning. *Knowledge-Based Systems* 191 (2020), 105233.

[57] Yin Zhu, Yuqiang Chen, Zhongqi Lu, Sinno Jialin Pan, Gui-Rong Xue, Yong Yu, and Qiang Yang. 2011. Heterogeneous transfer learning for image classification. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.