AMF-Placer: High-Performance Analytical Mixed-size Placer for FPGA

Tingyuan Liang*, Gengjie Chen†, Jieru Zhao‡, Sharad Sinha§ and Wei Zhang*
*ECE Department, Hong Kong University of Science and Technology; †CSE Department, Chinese University of Hong Kong
‡CSE Department, Shanghai Jiao Tong University; §CSE Department, Indian Institute of Technology Goa
tliang@connect.ust.hk, gjchen@cse.cuhk.edu.hk, zhao-jieru@sjtu.edu.cn, sharad@iitgoa.ac.in, eeweiz@ust.hk

***Abstract*—To enable the performance optimization of application mapping on modern field-programmable gate arrays (FPGAs), certain critical path portions of the designs might be prearranged into many multi-cell macros during synthesis. These movable macros with constraints of shape and resources lead to challenging mixed-size placement for FPGA designs which cannot be addressed by previous works of analytical placers. In this work, we propose AMF-Placer, an open-source Analytical Mixed-size FPGA placer supporting mixed-size placement on FPGA, with an interface to Xilinx Vivado. To speed up the convergence and improve the quality of the placement, AMF-Placer is equipped with a series of new techniques for wirelength optimization, cell spreading, packing, and legalization. Based on a set of the latest large open-source benchmarks from various domains for Xilinx Ultrascale FPGAs, experimental results indicate that AMF-Placer can improve HPWL by 20.4%-89.3% and reduce runtime by 8.0%-84.2%, compared to the baseline. Furthermore, utilizing the parallelism of the proposed algorithms, with 8 threads, the placement procedure can be accelerated by 2.41x on average.**

***Index Terms*—analytical placement, mixed-size placement, FPGA**

I. Introduction

With advancement in semiconductor technology, field-programmable gate arrays (FPGAs) have increased in size as well as the variety of resources available on them. This brings new challenges to the FPGA mapping flow.

A. Background and Previous Works

As shown in the example in Fig. 1, the latest island-style FPGA contains a 2-D array of configurable sites, each of which consists of basic elements of logic (BELs) [1]. For example, configurable logic block (CLB) sites consist of BELs like look-up tables (LUTs), flip-flops (FFs), multiplexers (MUXs) and carry chains (CARRYs). Some other sites contain larger heterogeneous BELs, e.g, digital signal processors (DSPs) and block random access memories (BRAMs). During FPGA placement, the instances in the netlist generated by logic synthesis should be placed on discrete sites on the FPGA device, with the goal of shorter routing wirelength, less congestion regions and better timing, under the constraints of the device architecture. To realize high-quality FPGA placement with high efficiency, there are three major challenges. First, the netlists of FPGA designs consist of heterogeneous instances, which require different hardware resources and should be placed in their specific legal discrete sites. Second,

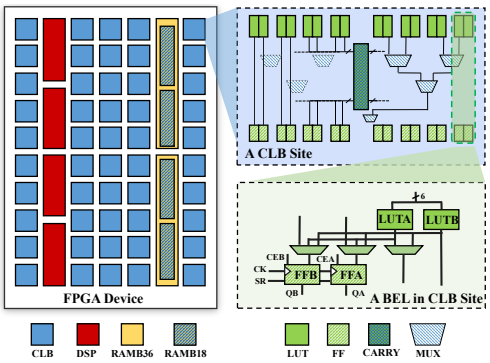


Fig. 1. Example of Xilinx Ultrascale FPGA device, a CLB site, and a BEL in it

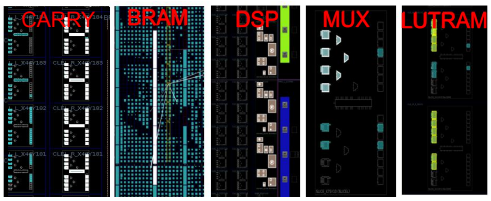


Fig. 2. Example of various types of macros with shape constraints: the macros are highlighted.

with the size of designs raised to the level of more than 1M instances, the scalability of the placer runtime becomes critical. Third, due to the upstream optimization, macros with constraints of shape and resource [2] [3] might be generated, like the examples shown in Fig. 2. In this paper, we use **standard cell** to denote the smallest, indivisible, representable component in the design netlist and use **macro** to denote a fixed group of multiple standard cells with constraints of their relative locations. For example: (1) 1 MUX and 2 LUTs connected to it should be treated as a macro; (2) a BRAM, without cascading with other BRAMs, is a standard cell; (3) 3 cascaded DSPs should be regarded as a macro. On the FPGA device, a macro might require multiple BELs spanning sites. Moreover, macros will lead to high interconnection density. These scenarios are seldom considered in previous exploration of FPGA placement.

Some FPGA placers [4] [5], e.g., VTR, are based on simulated annealing (SA), which might lead to long synthesis time when the input netlist is large. Thus, analytical solutions using numerical approaches were proposed to solve the placement with high scalability and quality [6]. Gort and Anderson [6]

presented an analytical FPGA placer HeAP, which demonstrated a 7.4× runtime advantage with 6% better placement quality compared to the SA placement algorithm of VPR 5.0. Chen et al. [7] proposed analytical placement solution with efficient and effective packing achieves 50% shorter wirelength, with an 18.30× overall speedup compared to VPR 7.0. During ISPD 2015/2016 contest, a series of analytical placers, e.g., UTPlaceF [8], RippleFPGA [9] and GPlace [10], were inspired with the consideration of congestion and clock constraints and they showed promising performance on the contest benchmarks. Later in 2017, LIQUID [11] was proposed with analytical solution based on gradient-guided algorithm while elfPlace [12] cast the placement density cost to the potential energy of an electrostatic system which tried to include various cost metrics in one nonlinear model to be optimized.

B. Challenges of FPGA Analytical Mixed-Size Placement

However, most of previous FPGA analytical placers except HeAP [6] targeted at benchmarks which consist of only standard cells, each of which will only occupy one BEL on FPGA device, e.g., the randomly generated benchmarks in ISPD 2015/2016 contest [13] [14]. The aforementioned macros with shape constraints are not considered by existing FPGA analytical placers proposed previously and this leads to challenges in wirelength optimization, cell spreading, packing and legalization during placement:

- Compared to standard cells, macros have many more pins and nets connected with other instances. For example, a CARRY macro, which might consists of 27 LUT cells, 25 FF cells and 3 CARRY cells, could connect to more than 200 nets outside the macro. During wirelength optimization with wirelength estimation model, a slight movement of these macros might lead to large distortion of the estimation.
- Some macros in FPGA design could share CLB with other instances at fine-grained level. It means that the macros and standard cells can "overlap" to some extents in FPGA mixed-size placement.
- Legalization in previous works on FPGA analytical placement are 1-to-1 legalization, i.e., one instance will require only one site. However, in mixed-size placement, the legalization of macros might be 1-to-many, i.e., one instance might require multiple sites. Moreover, the existing one-by-one iterative greedy legalization might be trapped in local optimum and lead to low efficiency.
- Latest designs might include a large number of macros. For example, in Minimap2 [15], there are 8782 macros and 499104 standard cells. In comparison, each of the benchmarks with macros in [6] has less than 54748 instances and less than 210 macros. Existing cell spreading algorithms for FPGA might fail to efficiently resolve resource overflow caused by a large number of macros.

With the consideration of the scenarios in real applications where there are elements with shape constraints, the characteristics of AMF-Placer are highlighted as follows:

- enable the mixed-size placement of large-scale design with macros of various types for FPGA-based designs.
- optimize the phases of placement especially for FPGA mixed-size placement, including
  - initial placement based on simulated-annealing
  - additional pseudo nets for efficient legalization and interconnection-density-aware weights of pseudo nets for quadratic placement to boost high-quality convergence
  - utilization-guided search of spreading window of overflowed region, resource supply fluctuation injection and location update with forgetting rate for cell spreading
  - progressive macro legalization
- parallelize the algorithm for each stage of the placement to reduce runtime.
- evaluate the placement quality and runtime with latest open-source large FPGA benchmarks from various application domains.

The source code and Wiki of our proposed AMF-Placer and involved open-source benchmarks are available at https://github.com/zslwyuan/AMF-Placer.

II. Preliminaries

In this section, we describe the mixed-size placement problem in FPGA scenarios and our analytical placement framework.

A. FPGA Macro Characteristics

As per examples shown in Fig.2, the standard cells in a macro must be placed in adjacent sites in the same column according to the downstream flow requirements. Usually, each macro could include one type of core cells, which could be CARRY cells, MUX cells, LUTRAM cells, DSP cells, or BRAM cells. Apart from the core cells, a macro might also include some peripheral LUT/FF cells, which are directly connected to the core cells. According to the core cell type, the macros can be mainly classified into 5 types and their major characteristics are listed as below:

- The CARRYs connected with carry in/out port should be extracted as a macro and moreover, the LUTs and FFs connected to the related CARRYs should be assigned in the same macro. Furthermore, to enable the routing of some input pins of CARRY, which connect to signals outside the CLB site, some corresponding LUT slots in the same CLB site should be transformed into non-logic route-thru LUTs, which are not in the original netlist. Similarly, FF slots in CLBs might be unavailable due to routing resource contention in CARRY macros.
- A MUX with its two input standard cells, which could be two LUTs or two other MUXes, should be extracted as a macro. MUX macros will also lead to route-thru usage of LUTs or disable external interconnection of some FFs because of the routing of selection signals.
- LUTRAM standard cells, which share input net for read/write address and data bits, should be extracted as

a macro. This kind of macros have to be located in SLICEM columns of the device.
- For DSPs and RAMs, they might be cascaded to handle larger demand of computation and storage. The standard cells in one of these macros are interconnected by the nets of their cascaded input/output signals. Please note that for each RAMB36E2 standard cell, we will consider it as a macro with 2 RAMB18E2 for legalization.

AMF-Placer automatically detects the macros in design netlist and generates virtual LUTs/FFs/MUXs to occupy resources and meet internal routing constraints. There are some other minor macros defined by vendor primitives [3], which are out of the scope of this work. More details are available in the device documentations [16]–[18].

B. Problem Formulation

The placement of the instances in a FPGA-based design can be formulated as a hypergraph $H = (V, E)$ placement problem. Let vertices $V = \{v_1, v_2, \ldots, v_n\}$ represent n instances in the design netlist and hyperedges $E = \{e_1, e_2, \ldots, e_m\}$ represent m nets. Let x_i and y_i be the x and y coordinates of the center of the instance v_i during placement, respectively. As mentioned in Section I-A, the instances can be categorized into two types, i.e., standard cells and macros, and both of these two types could be movable or fixed according to the design constraints. The most common objective function for placement is the sum of half-perimeter wirelength (HPWL) over all nets, i.e., the defined E. The FPGA mixed-size placer should determine the position of each movable instance (i.e., x_i and y_i) so that the total HPWL of the nets is minimized under the technology and region constraints.

C. The Framework of AMF-Placer

The workflow of AMF-Placer is shown in Fig. 3. The input of AMF-Placer is the pre-implementation netlist extracted from Xilinx Vivado and the output is the location of each instance on the specific device. The proposed placement consists of 7 phases as follows:

1) Initial Placement: Different from previous FPGA analytical placers, which start from random placement of instances, AMF-Placer starts from the initial placement generated by the SA-based placement of the instance clusters, utilizing the random factors in SA to overcome the limitation of analytical placers.

2) Quadratic Placement: For each instance, its next location will be determined by solving a quadratic optimization problem of HPWL, for wirelength minimization. To handle mixed-size placement, interconnection density and legalization are considered in quadratic placement with proposed adaptive pseudo net weights and pseudo nets for legalization.

3) Cell Spreading: Based on the area demand of instances and the area supply of the devices, the instances will be spread from the regions where demand for resources is outrunning supply, to other regions. It realizes the goal of rough legalization of fine-grained instances. To handle macros spanning large regions, a set of new techniques are involved.

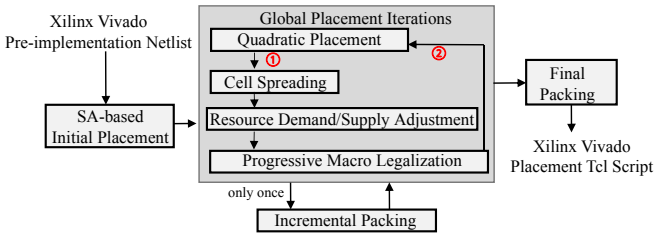


Fig. 3. The Outline of AMF-Placer's Workflow with 7 Process Phases: Label 1 indicates lower-bound placement and label 2 indicates upper-bound placer.

4) Resource Demand/Supply Adjustment: Based on the packing feasibility and routing congestion level, the area demand of some standard cells like LUTs and FFs will shrink or increase and the area supply of some regions will be increased or reduced, to improve the placement quality.

5) Progressive Macro Legalization: In our proposed flow for FPGA mixed-size placement, each macro will be mapped to multiple potential locations or one exact location according to the confidence. Moreover, anchors will be set to those locations for the macro and pseudo nets will be inserted between the macro and the anchors to facilitate the convergence.

6) Incremental Packing: During the global placement, at the fine-grained level, some LUTs and FFs will be paired as LUT-FF macros and FF-FF macros to shrink the problem size and improve the placement quality by identifying CLB internal nets at early stage.

7) Final Packing: After the global placement iterations, each instance will be mapped to sites, each of which consists of fixed number and types of resource, as final legalization or detailed placement. For example, LUTs, FFs, MUXs and CARRYs should be mapped to CLB.

The pairing algorithm in incremental packing is adopted from RippleFPGA [9]. Mechanisms of resource demand/supply adjustment and final packing are adopted from extended UTPlaceF [19], with modifications to support mixed-size placement.

As shown in Fig. 3, quadratic placement will generate lower-bound placement with lower HPWL (i.e., $\text{HPWL}_{\text{lower}}$). Correspondingly, the placement before quadratic placement with higher HPWL (i.e., $\text{HPWL}_{\text{upper}}$) is called upper-bound placement. When difference between upper-bound placement and lower-bound placement is close and macros are legalized exactly, the placement procedure will converge. All the related algorithms for these 7 phases are parallelized. Detailed methodologies will be illustrated in the following sections.

III. Implementation of AMF-Placer

A. Initial Placement

In most of the previous FPGA analytical placers, the initial placement is generated randomly, since it is claimed that the analytical placers are usually insensitive to the initial placement. However, most of them are evaluated on randomly generated FPGA netlists [13] [14] or small designs [6].

On FPGA, the resources are separated into discrete regions and the resource supply for each type of resource is not

even on the overall device. In the analytical models for these scenarios, the existences of local energy minima are obvious and this is a common limitation of analytical solutions.

Therefore, to overcome the limitation of existing analytical solutions, we proposed SA-based cluster-level initial placement before global placement iterations. AMF-Placer first recursively bi-partitions the input netlist H into clusters $C = \{c_1, c_2, \ldots, c_n\}$, each of which consists of at most N_{Csize} standard cells (including those in macros), based on PaToH [20], which can achieve runtime complexity of nearly $O(m)$. Since the partitioning procedures of the separated clusters are independent, they can be assigned to different threads to speed up the overall partitioning procedure. Moreover, we also set resource constraints, e.g., the number of DSPs, for the clusters to ensure each of them will not lead to serious resource overflow after initial placement. Meanwhile, we adopt the clock-aware partitioning criteria from [21] to generate a clock-friendly initial placement.

After partitioning, the FPGA device will be evenly divided into a grid of $N_{CY} \times N_{CX}$ bins $B_C = \{B_{Cij}\}$. For AMF-Placer, N_{CY} and N_{CX} are empirically set to be 8 and 5. Simulated-annealing (SA) algorithm is used to lower the cost function by randomly assigning the clusters to the bins and swapping them. Assuming all the instances in a cluster will be placed at the center of the bin where the cluster is assigned, the cost function F_{cost}, consisting of the terms for wirelength and instance density, is defined as follows:

$$F_{cost} = \sum_{e \in E} W_e + \sum_{B_{Cij} \in B_C} Of(B_{Cij}) \tag{1}$$

$$Of(B_{Cij}) = \frac{(W_B + H_B)(\sum_{c \in B_{Cij}} size(c))^2}{N_{Csize}} \tag{2}$$

where W_e is the HPWL of net e and $Of(B_{Cij})$ is the resource overflow function of the bin B_{Cij} since multiple clusters could be assigned to it. W_B and H_B are the width and height of a cluster bin. With random factors, instances can realize movement in a wide range and the limitations of resource supply constraints, which might trap instances in local optima, can be overcome. Furthermore, to enlarge the search space of the algorithm, the SA procedure will run multiple times with different random start points. These SA procedures with different start points can be parallelized as well.

B. Quadratic Placement

As mentioned in Section I-A, analytical placers approximate the wirelength (or HPWL) and some other metrics in numerical models for efficient solutions. In this paper, like what many previous analytical placers [6] [8] [9] [22] [19] did, to approximate the underivable HPWL function,

$$W(\mathbf{x}, \mathbf{y}) = \sum_{e \in E} W_e = \sum_{e \in E} (\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j|), \tag{3}$$

AMF-Placer use weighted quadratic objective function $\widetilde{W_e}$, as follows:

$$\widetilde{W_e} = \sum_{i,j \in e} [w_{x,ij}^{B2B}(x_i - x_j)^2 + w_{y,ij}^{B2B}(y_i - y_j)^2], \tag{4}$$

where $w_{x,ij}^{B2B}$ and $w_{y,ij}^{B2B}$ are weights set according to Bound2Bound net model [23]. Then, we can formulate the global placement problem for wirelength minimization as a constrained minimization problem as follows:

$$\begin{aligned} \min_{\mathbf{x},\mathbf{y}} \quad & \sum_{e \in E} \widetilde{W_e} \\ \text{s.t.} \quad & x_i, y_i \text{ is legal for the type of instance } v_i \end{aligned} \tag{5}$$

Here, a location of an instance is legal when there are enough specific resources for the type of the instance at that location. To make this constrained problem solvable by a quadratic solver, virtual anchors are added to the model. These virtual anchors will be connected to their corresponding instances with artificial two-pin pseudo nets to guide the instances to the legal locations. Moreover, the values of $w_{x,ij}^{B2B}$ and $w_{y,ij}^{B2B}$ are determined by the current overall placement, i.e., $\mathbf{x}$ and $\mathbf{y}$. If the placement gets significant change, $\widetilde{W_e}$ might poorly approximate the actual HPWL. Therefore, pseudo nets can limit the movement of instances so the placer can update $w_{x,ij}^{B2B}$ and $w_{y,ij}^{B2B}$ between quadratic placement iterations. Finally, the problem can transform into one without constraints:

$$\min_{\mathbf{x},\mathbf{y}} \quad \sum_{e \in E} \widetilde{W_e} + \sum_{e_p \in E_p} [w_{e_p} \widetilde{W_{e_p}}] \tag{6}$$

where $E_p = \{e_p\}$ is the set of pseudo nets, w_{e_p} is the extra specific weight for pseudo net and $\widetilde{W_{e_p}}$ is the HPWL quadratic approximation of e_p. An example in Fig. 4 shows that an instance could connect to multiple pseudo nets. The greater w_{e_p} is, the more closely the instance can move around from the anchors when solving the quadratic problem.

The insertion of pseudo nets and their weights w_{e_p} can significantly impact the quality and runtime of placement. In [9], [22], [24] and [11], there are only pseudo nets interconnecting instances with their locations in last quadratic placement iteration, i.e., the red dash lines in Fig. 4.

First, since available sites for the legalization of large macros without overlap are sparse on FPGA, simply considering wirelength minimization without legalization objective during quadratic program will worsen the wirelength after final legalization. To facilitate macro legalization in mixed-size placement, AMF-Placer inserts additional pseudo nets interconnecting instances with their several potential legal locations. The way to find these locations and the reason why the additional pseudo nets help are explained in Section III-D.

Second, in previous works, the weight w_{e_p} of a pseudo net is calculated by dividing a global factor α by the movement distance of the corresponding instance in last optimization

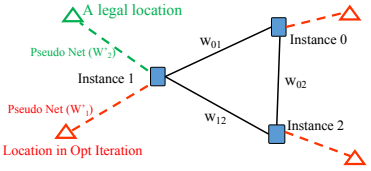


Fig. 4. An Example of Pseudo Nets and Anchors: the dash lines indicate pseudo nets and the triangles represent anchors.

iteration, like in Eqn.7. By gradually increasing the value of α, the global placement will tend to converge.

$$w_{e_p,v_i} = \alpha/\text{movement}(v_i) \quad (\text{[9], [22] [24] and [11]}) \quad (7)$$

$$w_{e_p,v_i} = \alpha/\text{movement}(v_i) \times \text{pinNum}(v_i) \quad (\text{ours}) \quad (8)$$

However, considering macros, a wide range of movements of instances connecting to many nets will lead to serious distortion of the wirelength estimation in Eqn. 4. This will make the placement convergence procedure highly unstable when there are a large number of macros in the design. Therefore, AMF-Placer adds an extra multiplier factor for each pseudo net, which is the number of external pins for the corresponding instance, as shown in Eqn.8. With such interconnection-density-aware setting, the macros will be moved slower than the other smaller instances and have heavier gravity toward their potential legal positions, facilitating faster convergence to lower HPWL. Finally, the Eigen3 solver [25] is adopted to handle the optimization in Eqn.6 with the high parallelism based on OpenMP.

C. Cell Spreading

For a specific region on FPGA, the available resources of various types are limited. Therefore, the instances should be spread from the regions where demand for resources is outrunning supply, to other regions, as rough legalization. The fundamental cell spreading algorithm of AMF-Placer is based on widely adopted bi-partitioning rough legalization [4] [6] [9] [22] [26] [11]. During cell spreading, the FPGA device will be evenly divided into a grid of $N_Y \times N_X$ bins $B = \{B_{ij}\}$. The placer will find an overflowed bin, expand it into a corresponding larger window containing it, recursively partition the window, and spread the instances into bins in the window.

However, as mentioned in Section II-A, macros might require high density of resource, span multiple sites and share

Algorithm 1: AMF-Placer Cell Spreading

Input: instance locations in lower-bound placement $\mathbf{P}^L = \{(x_i^L, y_i^L)\}$, instance locations in last upper-bound placement $\mathbf{P}^U = \{(x_i^U, y_i^U)\}$, forgetting rate γ

Output: instance locations in new upper-bound placement $\mathbf{P}^{U'} = \{(x_i^{U'}, y_i^{U'})\}$

1 OFBins = findAndSortOverflowBins($\mathbf{P}^L$);
2 OFCells = cells in OFBins;
3 resetOverflowTimes(OFBins);
4 **while** $size(OFCells) > \theta_{of}$ **do**
5 updateOverflowTimes(OFBins);
6 injectSupplyFluctuation(OFBins);
7 nonOverlapWins = greedilyFindNonOverlapWindow(OFBins);
8 **parallel foreach** *Win in nonOverlapWins* **do**
9 $\mathbf{P}'_{\text{cell}}$ = cellSpreading(Win); // update locally
10 $\mathbf{P}'$ = updateMacroLocation($\mathbf{P}'_{\text{cell}}$);
11 $\mathbf{P}^{U'}$ = updateLBPlacementWithγ($\mathbf{P}'$);
12 OFBins = findAndSortOverflowBins($\mathbf{P}^{U'}$);
13 OFCells = cells in OFBins;
14 adjustResourceDemandSupply();

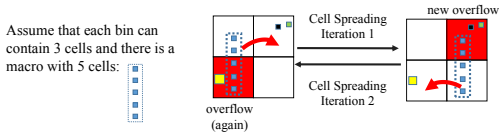


Fig. 5. An Example of Cell Spreading Deadlock of Macro

CLB sites with standard cells, so AMF-Placer includes new methods to handle such mixed-size spreading, as illustrated in Algorithm 1. Details are explained as follows.

As shown in Fig. 5, macros might cause deadlocks in the common partitioning-based cell spreading algorithm because moving a macro spanning multiple bins from an overflowed region might cause new overflows in other regions. AMF-Placer's solution is to inject resource fluctuation to those bins trapped in deadlocks(line 5 and 6 in Algorithm 1), i.e., periodically reduce and recover the resource supply of those bins, and squeeze out some instances from them. The extent of supply fluctuation for a bin is proportional to the number of times the bin gets overflowed. The fluctuation will stop immediately when there is no overflow.

To find a minimal rectangular window for spreading from an overflowed bin, [26] conducted an enumeration search, which could be time-consuming, while [9] expanded the window from the overflowed bin with a pre-determined expanding pattern, which might lead to unnecessary window expanding and undermine wirelength. In contrast, AMF-Placer will iteratively expand the window. For each expanding iteration, it will first check the resource utilization of the neighbor bins in four directions. Then, to expand the window, it will select the direction, where the neighbor bins have the lowest utilization compared to the other directions (line 7 in in Algorithm 1). This iterative procedure for an overflowed bin will continue until the window covers enough resource.

To enable parallelized cell spreading, AMF-Placer will sort the overflowed bins according to their resource utilization. Those bins with higher utilization will have higher priority to find their spreading regions. During the speading window expanding for an overflowed bin, the bins in the window will be colored. The later window expanding for the other overflowed bins cannot cover the colored bins. The obtained windows will not overlap with each others and can conduct standard-cell-level spreading concurrently (line 8 in Algorithm 1).

For a rectangular cell spreading window for an overflowed bin with sufficient resources, [4] and [11] required it to completely cover the macros inside it, which will lead to over-spreading and high HPWL when there are many large macros close to each other. To solve this problem, as inspired by ASIC mixed-size placer [27], AMF-Placer conducts two-phase cell spreading:

- In the first phase, standard cells in macros will be released from the shape constraints and spread with the other common standard cells to resolve the resource overflow.
- In the second phase, a macro's location will be updated to the average location of the standard cells in it.

By iteratively involving the two phases (line 9-10 in Algo-

rithm 1), resource overflows will be gradually resolved.

Finally, the partitioning-based cell spreading algorithm is sensitive to the overflow. When a placement is compact, a single overflowed bin could cause cell spreading in a large region and seriously increase wirelength. Therefore, instead of directly updating instance location, we keep the location of an instance v_i in last upper-bound placement as (x_i^U, y_i^U). During cell spreading, when the instance is expected to spread to the location (x_i', y_i'), the actual updated location will be set to:

$$(\gamma x_i' + (1-\gamma)x_i^U, \gamma y_i' + (1-\gamma)y_i^U)$$

where γ is forgetting rate and set to be $(1 - 0.95 \times \text{HPWL}_{\text{lower}}/\text{HPWL}_{\text{upper}})$ in AMF-Placer (line 11 in Algorithm 1). It can gradually depress the sensitivity of cell spreading as the placement tends to converge.

After each cell spreading iteration, the resource demand and supply will be adjusted according to packing and routing feasibility [19].

D. Progressive Macro Legalization

As discussed in Section I-B, 1-to-1 legalization cannot handle macro legalization. To solve this challenge of mixed-size FPGA placement, we propose progressive macro legalization for the 1-to-many legalization of a large number of macros in AMF-Placer. AMF-Placer will only conduct rough legalization in the early iterations of global placement, and it will conduct exact legalization following rough legalization when the macros are close enough to their potential legal positions. The overall flow is shown in Algorithm 2.

Corresponding to line 1-14 in Algorithm 2, during rough legalization of macros, the standard cells in the macros will be released from the shape constraints and legalized individually. According to a given displacement threshold θ_{disp}, a set of candidate sites $S_{rs} = \{s_r\}$ will be found for each standard cell in the macro (line 5 in Algorithm 2). A bipartite graph for the mapping between standard cells and candidate sites will be constructed (line 6 in Algorithm 2). In this bipartite graph, the weight of an edge is the HPWL increase when the cell moves to the corresponding candidate site indicated by the edge. Then, min-cost bipartite matching will map each standard cell to an FPGA site. As the macros spread over the device, the bipartite graph might consist of multiple independent connected subgraphs. Accordingly, the matching procedures for these subgraphs can be fully parallelized (line 7-8 in Algorithm 2). If there are some standard cells which cannot match with site, θ_{disp} will be increased by $\Delta\theta_{disp}$ to include more unmatched candidate sites for the unmatched cells and the matching algorithm will be rerun for them. θ_{disp} will be increased iteratively until all cells are matched. After the matching, for each macro, a pseudo net will connect the macro to the average location of the mapped sites of the standard cells inside it (line 14 in Algorithm 2). In rough legalization, standard cells in a macro are allowed to be mapped to different columns. In the early stage of placement, the density of instances is high and their locations are highly unstable. Meanwhile, macros consisting of DSPs, LUTRAMs, or BRAMs are required to be mapped to the sparse legal regions and some of them might span tens of sites. In this situation, direct legalization [8] or cell spreading [9] for macros might lead to serious displacement and trap the placement in bad local optima. AMF-Placer's solution gradually strengthens the weights of the legalization pseudo nets, smoothly legalizes the large macros, and preserves space for wirelength optimization.

Algorithm 2: Progressive Macro Legalization

Input: upper-bound placement $\mathbf{P}^U = \{(x_i^U, y_i^U)\}$, candidate site number for each cell/macro N_{cand}, initial displacement threshod θ_{disp_I}, threshold increment $\Delta\theta_{disp}$ and displacement threshold to enable exact legalization θ_{exact}

Output: psuedo nets connected to legalization anchors for macros $E_p^{Legal} = \{e_p^{Legal}\}$

```
1  unmappedCells = findUnmappedCellsInMacros();
2  cell2SiteRoughMap = ∅;
3  θ_disp = θ_disp_I;
4  while size(unmappedCells) > 0 do
5      cell2CandidateSiteMap =
         findCandidateSites(unmappedCells, θ_disp, N_cand);
6      BPGraphs = createIndependentBipartite-
         Graphs(cell2CandidateSiteMap);
7      parallel foreach G in BPGraphs do
8          minCostBipartiteMatching(G);
9      updateRoughLegalization(cell2SiteRoughMap);
10     unmappedCells = findUnmappedCellsInMacros();
11     θ_disp = θ_disp + Δθ_disp;
12 avgDisp = checkAvgDisp(cell2SiteRoughMap);
13 if avgDisp > θ_exact then
14     E_p^Legal = linkToAvgLoc(cellSiteRoughMap)
15 else
16     macro2ColumnMap = spreadMacrosToCols(P^U)
17     parallel foreach col in Cols do
18         macros = getMacrosIn(col);
19         sortByY(macro);
20         macro2SiteExactMap = DPBasedLegalization(macro,
             col);
21         E_p^Legal = linkToExactLoc(macro2SiteExactMap);
```

As pseudo weights iteratively increase, the macros get closer to the legal locations. If the average displacement from cells to sites in rough legalization is lower than a threshold θ_{exact}, exact (strict) legalization will follow the rough legalization to ensure that the standard cells in a macro must be placed in adjacent sites in the same column. The workflow of exact legalization is demonstrated in line 16-21 in Algorithm 2. First, the cell-spreading-based approach [9] is utilized to map the macros to the resource columns, which provide resources of the corresponding type (line 16 in Algorithm 2). For intra-column legalization, suppose there are $Nmacro, col$ macros assigned to the column and $Nsite, col$ sites in the column. For macros in the column, they will be sorted according to their y-coordinates. Then they will be assigned indices, $0, \ldots, N_{macro,col} - 1$, from the bottom one to the top one. By ensuring their order in the vertical direction is unchanged, the intra-column macro legalization can be solved via dynamic

programming (DP):

$$f(i,j) = min(f(i-1, j-row(i)) + \text{HPWL}^{+}(i, j-row(i)+1), f(i,j-1)) \quad (9)$$

where $f(i,j)$ represents the increase of HPWL to legalize the 0-i th macros in 0-j th rows of the column, $row(i)$ represents the number of adjacent rows which should be occuplied by macro i, and HPWL^{+} denotes the HPWL increase when macro i is placed from row $j-row(i)+1$ to row j, compared to the placement before legalization. Variable i will iterate from 0 to $N_{macro,col}-1$ and Variable j will iterate from 0 to $N_{site,col}-1$. Constraints and initial state settings for Eqn.9 are not shown because of limited space. Since the macros are close to their potential legal positions and the placement tends to be stable at this stage, we assume that the locations of the other instances will remain unchanged during the intra-column legalization of macros. Therefore, the intra-column legalization procedures for different columns can be parallelized (line 17-21 in Algorithm 2).

E. Final Packing

For the final packing, a parallelized packing algorithm with high quality has been proposed by [19]. It allows the FPGA sites to search their corresponding candidate packing solutions concurrently and then negotiate together during synchronization. It handles exception instances that cannot be packed during the parallel method with the sequential conventional ripping-up algorithm. We adapt this high-performance algorithm to our mixed-size placement scenarios with several major modifications: (1) Some instances might be pre-packed into CLB sites by the macro legalization, but these pre-packed CLBs can be filled with other instances as long as the rules described in Section II-A are obeyed; (2) We encode each candidate packing cluster for a specific site with hash function to avoid many redundant packing attempts to improve performance and quality of packing; (3) By iteratively increasing the ripping-up window size and processing independent exception instances concurrently, we also parallelize the exception handling algorithm and achieve further acceleration.

IV. Experimental Results

A. Target Device, Benchmarks and Environment

Currently, AMF-Placer targets at Xilinx VU095 FPGA devices. We collect the latest large open-source benchmarks which are suitable for VU095 device and designed for various domains, including CNN [28], memory networks [29], LSTM [30], SoC [31], NoC [32], and genetic encode alignment [15]. Their parameters are listed in Table I, where we use macroRatio, the quotient of the total number of sites required by macros and the total number of sites required by all instances, to indicate the macro proportion of the design. Some of the benchmarks are generated via high-level synthesis [33] while the others are described in Verilog. Some of them contain commercial IP cores. These IP cores are black-box instances in the post-synthesis netlist and cannot be handled by RapidWright [34] and other placers relying on the EDIF (Electronic Design Interchange Format) netlists exported from commercial tools. To overcome this limitation, AMF-Placer directly extracts the instance interconnection from Vivado via interactive Tcl commands, to handle general designs. AMF-Placer is implemented in C++ and experiments are conducted on Ubuntu 20.04 with Intel i7-6770 CPU (3.40 GHz, 8 logic cores) and 32GB DDR4. Due to the license restriction of the Vivado patch in ISPD 2015/2016 contests, we cannot utilize the patch to get the routed wirelength from Vivado and we use HPWL to approximate the final routed wirelength.

TABLE I: Benchmark Parameters

Benchmarks	Rosetta FaceDetection	SpooNN	OptimSoC	MiniMap2	OpenPiton	MemN2N	BLSTM	Rosetta DigitRecog
#LUT	68945	63095	186183	407586	180388	184997	118967	151636
#FF	56987	70987	248983	252624	111966	84694	54690	105580
#CARRY	4978	2091	1715	19826	1712	11528	2762	1970
#Mux	2177	217	27037	180	13696	4466	36210	4662
#LUTRAM	255	251	901	251	752	3500	1147	251
#DSP	101	165	51	528	58	312	258	1
#BRAM	141	208	218	283	147	148	812	379
#Cell	134450	137937	468150	681889	309145	289721	215101	265775
#Macro	3582	1135	21882	8746	8278	5775	14651	3061
#siteForMacro	55666	23079	89004	191263	48066	118960	171822	55754
MacroRatio	40%	16%	19%	28%	15%	41%	80%	21%

B. Effectiveness of Proposed Optimization Techniques

In this subsection, we evaluate the impact of the proposed optimizations for different placement phases. Here we show their impact by disabling different specified optimization technique and these techniques include:

- *Tech1*: SA-based initial placement
- *Tech2*: interconnection-density-aware pseudo net weight
- *Tech3*: utilization-guided search of spreading window
- *Tech4*: forgetting-rate-based cell spreading update
- *Tech5*: progressive macro legalization

Since existing open-source analytical FPGA placers do not support mixed-size FPGA placement of aforementioned macros on Ultrascale devices, for comprehensive comparison, according to some state-of-the-art solutions, we implement baseline placement solution with the following features:

- quadratic placement, cell spreading and clock region planning algorithms from RippleFPGA [9] and clock-aware initial clustering from [21].
- resource demand adjustment and packing algorithms from extended UTPlaceF [19]
- SA initial placement
- necessary modifications to support macro placement, e.g., macro legalization/packing, but without *Tech2-5*
- parallelized

The experimental results are shown in Table II. Our proposed solution achieves the best HPWL compared to other configurations. Compared to baseline, AMF-Placer can improve HPWL by 20.4%-89.3% and reduce runtime by 8.0%-84.2%. Results of experiments without *Tech1* show a good initial placement is crucial for analytical placement, especially for designs with clear hierarchy like OptimSoC [31] and OpenPiton [32]. Therefore, we also enable SA-based initial placement for the baseline since we think it should be a necessary process. The experiments without *Tech2* or *Tech3* indicate that utilization-guided search of spreading window

TABLE II: COMPARISON OF RUNTIME AND HPWL WITH VARIOUS BENCHMARKS AND CONFIGURATION

	Rosetta FaceDetect [33]				SpooNN [28]				OptimSoc [31]				MiniMap2 [15]			
	HPWL	R_{hpwl}	time(s)	R_{time}	HPWL	R_{hpwl}	time(s)	R_{time}	HPWL	R_{hpwl}	time(s)	R_{time}	HPWL	R_{hpwl}	time(s)	R_{time}
proposed	446908	**1.000**	105	1.054	339764	**1.000**	129	**1.000**	1771538	**1.000**	466	1.000	1275346	**1.000**	558	**1.000**
w/o *Tech1*	479221	1.072	109	1.088	360567	1.061	132	1.023	9034564	5.100	871	1.870	5132073	4.024	1265	2.266
w/o *Tech2*	465208	1.041	131	1.315	491385	1.446	517	4.007	1841786	1.040	577	1.239	1790566	1.404	1019	1.826
w/o *Tech3*	487060	1.090	124	1.238	431386	1.270	240	1.857	1825819	1.031	543	1.164	1307249	1.025	626	1.121
w/o *Tech4*	450199	1.007	100	**1.000**	351649	1.035	131	1.015	2658863	1.501	525	1.126	1291565	1.013	561	1.005
w/o *Tech5*	511514	1.145	134	1.344	443765	1.306	138	1.067	1880431	1.061	546	1.172	1430330	1.122	708	1.268
baseline	794201	1.777	234	2.338	1865746	5.491	329	2.549	2231978	1.260	559	1.199	11886747	9.320	3539	6.339
	OpenPiton [32]				Rosetta DigitRecog [33]				MemN2N [29]				BLSTM [30]			
proposed	1139189	**1.000**	283	1.035	726929	**1.000**	254	1.097	801403	**1.000**	318	1.130	484650	**1.000**	326	1.241
w/o *Tech1*	6218009	5.458	350	1.280	768796	1.058	256	1.105	963416	1.202	363	1.287	492194	1.016	285	1.083
w/o *Tech2*	1159003	1.017	326	1.190	730071	1.004	321	1.389	916176	1.143	411	1.460	496540	1.025	364	1.383
w/o *Tech3*	1157479	1.016	296	1.079	773957	1.065	272	1.174	862212	1.076	343	1.216	511861	1.056	296	1.126
w/o *Tech4*	1212149	1.064	274	**1.000**	728094	1.002	231	**1.000**	822742	1.027	282	**1.000**	653010	1.347	307	1.169
w/o *Tech5*	1142927	1.003	301	1.098	997207	1.372	244	1.053	923670	1.153	353	1.252	722495	1.491	263	**1.000**
baseline	1432535	1.258	297	1.086	1047885	1.442	298	1.288	1610425	2.010	547	1.942	907383	1.872	325	1.237

TABLE III: ACCELERATION RATIOS WITH PARALLELISM

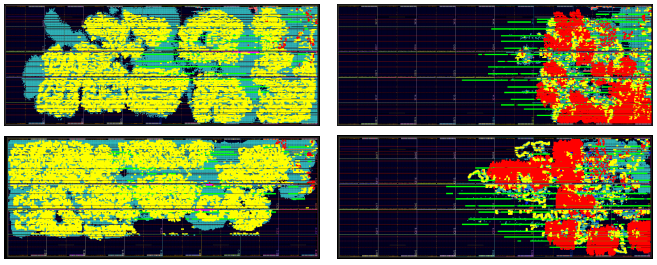
	Rosetta FaceDetect	SpooNN	OptimSoC	MiniMap2	OpenPiton	Rosetta DigitRecog	MemN2N	BLSTM
8 threads	2.17x	2.07x	2.50x	2.86x	2.63x	2.22x	2.61x	2.23x
4 threads	2.01x	1.96x	2.29x	2.57x	2.37x	2.04x	2.35x	1.97x
2 threads	1.56x	1.52x	1.64x	1.81x	1.65x	1.54x	1.68x	1.77x
1 threads	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x

and proposed pseudo net weight settings can improve the quality and runtime of placement. The experiments without *Tech4* can sometime achieve faster convergence by directly updating cells' locations in cell spreading, which might worsen HPWL when the placement is compact. Compared to the slight overhead of runtime, the advantage of *Tech4* is considerable. Moreover, the experiments without *Tech5* show that replacing progressive macro legalization with direct macro legalization [19] will undermine final wirelength, especially for SpooNN [28] and BLSTM [30], where there are many large BRAM macros. Finally, according to the experiments, the overall impact of the combination of the proposed techniques is noticeable. For example, in baseline, without *Tech3*, larger windows will be used to spread instances. In this situation, direct update of placement without *Tech4* will further increase the wirelength which can lead to large displacements of macros during legalization without *Tech5*. This is a vicious circle. Even worse, this bad global placement will make parallelized packing difficult and cause long runtime of packing.

As indicated in Section III, the dominant algorithm for each stage in the proposed placement flow can be parallelized and in Table III, acceleration ratios are demonstrated by changing the number of threads and evaluating placement runtime. Averagely, with 8 threads, 4 threads, and 2 threads, the placement can be accelerated by 2.41x, 2.20x and 1.64x respectively, compared to the single-thread placement. The marginal benefit of parallelism decreases as the thread number increases. The major reason is that the global placement iterations take up most of the runtime, and during cell spreading in global placement, as most of the overflowed regions are processed, the number of spreading windows, in line 8 of Algorithm 1, will decrease fast. As result, the space for parallelism will be limited if the number of threads is already large.

C. Portability to Commercial Tools

Placement generated by AMF-Placer can be loaded into Vivado to perform routing. All the placement generated from proposed placement flow can be successfully routed. However, we cannot get the routed wirelength from Vivado because of the license restriction of the Vivado patch in ISPD 2015/2016



(a) MiniMap2 (b) BLSTM

Fig. 6. Comparison of AMF-Placer Placement (upper ones) and Vivado Placement (lower ones): yellow for CARRY macros, red for MUX macros, green for BRAM macros, purple for DSP macros, blue for LUTRAM macros. The view of device is rotated left by 90°.

contests. The placement of the largest benchmark, MiniMap2 [15], and the benchmark with the highest macro ratio, BLSTM [30], are shown in Fig. 6 as examples. Different types of macros are highlighted in different colors. Due to limited space, more detailed statistics are available in the open-source project documentation. The difference of congestion extent between Vivado and AMF-Placer is small. The slice utilization of AMF-Placer is slightly higher than Vivado because Vivado's placement includes better floorplanning and utilizes design hierarchy information. Moreover, since Vivado placer is timing-driven while AMF-Placer is wirelength-driven, the current WNS/TNS results of AMF-Placer are worse than the Vivado placement results. Solutions to these problems will be parts of our future works to extent AMF-Placer.

V. CONCLUSION

In this work, we propose AMF-Placer, an open-source FPGA analytical placer supporting the FPGA mixed-size placement. To speed up the convergence and improve the quality for the mixed-size placement, AMF-Placer is equipped with a series of new parallelizable optimization techniques in quadratic placement, cell spreading, packing, and legalization. Based on the latest large open-source designs from various domains for Xilinx Ultrascale FPGAs, experimental results indicate that AMF-Placer can improve HPWL by 20.4%-89.3% and reduce runtime by 8.0%-84.2%, compared to the baseline. The source code and Wiki of AMF-Placer and involved open-source benchmarks are available at https://github.com/zslwyuan/AMF-Placer. We sincerely appreciate the kindly suggestions from reviewers, detailed explanations of UTPlaceF [19] from Dr. Wuxi Li, and useful advice on Vivado metric usages from Dr. Stephen Yang [13].

REFERENCES

[1] Xilinx, "Ug974: Ultrascale architecturelibraries guide," 2020.

[2] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Hmflow: Accelerating fpga compilation with hard macros for rapid prototyping," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 117–124.

[3] C. M. Fuller, S. W. Gould, S. P. Hartman, E. E. Millham, and G. Yasar, "Field programmable gate arrays using semi-hard multicell macros," Jun. 2 1998, uS Patent 5,761,078.

[4] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker *et al.*, "Vtr 8: High-performance cad and customizable fpga architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.

[5] G. Chen and J. Cong, "Simultaneous placement with clustering and duplication," in *Proceedings of the 41st annual Design Automation Conference*, 2004, pp. 740–772.

[6] M. Gort and J. H. Anderson, "Analytical placement for heterogeneous fpgas," in *22nd international conference on field programmable logic and applications (FPL)*. IEEE, 2012, pp. 143–150.

[7] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang, "Efficient and effective packing and analytical placement for large-scale heterogeneous fpgas," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2014, pp. 647–654.

[8] W. Li, S. Dhar, and D. Z. Pan, "Utplacef: A routability-driven fpga placer with physical and congestion aware packing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 4, pp. 869–882, 2017.

[9] C.-W. Pui, G. Chen, W.-K. Chow, K.-C. Lam, J. Kuang, P. Tu, H. Zhang, E. F. Young, and B. Yu, "Ripplefpga: A routability-driven placement for large-scale heterogeneous fpgas," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8.

[10] R. Pattison, Z. Abuowaimer, S. Areibi, G. Grewal, and A. Vannelli, "Gplace: A congestion-aware placement tool for ultrascale fpgas," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–7.

[11] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt, "Liquid: High quality scalable placement for large heterogeneous fpgas," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 17–24.

[12] W. Li, Y. Lin, and D. Z. Pan, "elfplace: Electrostatics-based placement for large-scale heterogeneous fpgas," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[13] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal, "Routability-driven fpga placement contest," in *Proceedings of the 2016 on International Symposium on Physical Design*, 2016, pp. 139–143.

[14] D. Stroobandt, J. Depreitere, and J. Van Campenhout, "Generating new benchmark designs using a multi-terminal net model," *Integration*, vol. 27, no. 2, pp. 113–129, 1999.

[15] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 127–135.

[16] Xilinx, "Ug574: Ultrascale architecture configurable logic block," 2017.

[17] ——, "Ug573: Ultrascale architecture memory resources," 2021.

[18] ——, "Ug579: Ultrascale architecture dsp slice," 2020.

[19] W. Li and D. Z. Pan, "A new paradigm for fpga placement without explicit packing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2113–2126, 2018.

[20] Ü. V. Çatalyürek and C. Aykanat, "Patoh (partitioning tool for hypergraphs)," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1479–1487.

[21] J. Chen, Z. Lin, Y.-C. Kuo, C.-C. Huang, Y.-W. Chang, S.-C. Chen, C.-H. Chiang, and S.-Y. Kuo, "Clock-aware placement for large-scale heterogeneous fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 5042–5055, 2020.

[22] M.-C. Kim, D.-J. Lee, and I. L. Markov, "Simpl: An effective placement algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 50–60, 2011.

[23] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Kraftwerk2a fast force-directed quadratic placement approach using an accurate net model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1398–1411, 2008.

[24] T. Lin and C. Chu, "Polar 2.0: An effective routability-driven placer," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[25] G. Guennebaud, B. Jacob *et al.*, "The eigen 3 c++ library," 2010.

[26] T. Lin, C. Chu, and G. Wu, "Polar 3.0: An ultrafast global placement engine," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 520–527.

[27] M.-C. Kim and I. L. Markov, "Complx: A competitive primal-dual lagrange optimization for global placement," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 747–752.

[28] K. Kara, "Spoonn: Fpga-based neural network inference library," 2018.

[29] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, "End-to-end memory networks," *arXiv preprint arXiv:1503.08895*, 2015.

[30] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "Hardware architecture of bidirectional long short-term memory neural network for optical character recognition," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1390–1395.

[31] S. Wallentowitz, P. Wagner, M. Tempelmeier, T. Wild, and A. Herkersdorf, "Open tiled manycore system-on-chip," *arXiv preprint arXiv:1304.5081*, 2013.

[32] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, "Openpiton: An open source manycore research framework," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 217–232, 2016.

[33] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.

[34] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 133–140.

[35] Y. Zhou, D. Vercruyce, and D. Stroobandt, "Accelerating fpga routing through algorithmic enhancements and connection-aware parallelization," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 4, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3406959