UTPlaceF 3.0: A Parallelization Framework for Modern FPGA Global Placement

(Invited Paper)

Wuxi Li, Meng Li, Jiajun Wang, and David Z. Pan
ECE Department, University of Texas at Austin, Austin, Texas USA
{*wuxili, meng_li, jiajunwang*}*@utexas.edu; dpan@ece.utexas.edu*

***Abstract*—Global placement is a major runtime bottleneck of modern FPGA physical synthesis. As the FPGA capacity grows rapidly, new innovative global placement approaches are in great demand for more efficient circuit mapping and prototyping. In this paper, we propose a parallelization framework for modern FPGA global placement, UTPlaceF 3.0. Two major techniques are presented to boost the performance of a state-of-the-art quadratic placer with only small quality degradation: 1) placement-driven block-Jacobi preconditioning and 2) parallelized incremental placement correction. Experimental results show that UTPlaceF 3.0 can take full advantages of modern multi-core CPUs and achieves more than 5X speedup over sequential implementation with competitive placement quality.**

I. Introduction

Placement is one of the most time-consuming optimization steps in modern FPGA physical synthesis flow. In the past several decades, most of the research attention and endeavor was focused on improving placement solution quality. However, ultra-fast and efficient placement core engines are now in great demand. Firstly, with the increasing capacity and complexity of modern FPGA devices, the gate count of state-of-the-art commercial FPGAs has reached the scale of millions [1], [2]. Moreover, as a type of hardware accelerators, FPGAs need to be frequently reconfigured to adapt rapid and continuous changes from users in many scenarios, like datacenter-based cloud computing. Therefore, it is particularly desirable to develop a high-performance placement engine to reduce the FPGA synthesis time. Besides, placement also has significant impacts on the quality of design mapping, hence, good performance-boosting techniques should still maintain competitive placement solution quality.

The scalability issue has constantly pushed the evolution of placement algorithms in the past few decades. In the early age of placement, simulated annealing-based placers, such as [3]–[5], dominated industry and academic research. Although the annealing technique worked well for small designs, as the capacity of FPGAs kept growing, it was no longer scalable for larger FPGA devices. Industry and academia then turned to min-cut-partitioning-based placers, e.g. [6], [7], which performed well for designs with tens of thousands gates. When the gate count of FPGAs arrived at the scale of millions, analytical placers, e.g. [8]–[16] started to steadily outperform min-cut-partitioning-based approaches in both runtime and quality.

Despite the effectiveness and efficiency of analytical placers, their runtime still increases rapidly as the complexity and scale of FPGA devices has not stopped growing. One promising solution is leveraging today's powerful multi-core CPUs to achieve efficient placement parallelism. For quadratic placers, wirelength is approximated as a quadratic objective, which can be minimized by solving symmetric and positive-definite (SPD) linear systems. The solution can be quickly approached by various Krylov-subspace methods, among which Preconditioned Conjugate Gradient (PCG) method is the most efficient known algorithm for placement problem. As solving SPD linear systems dominates the total runtime of placement, some previous effort has been made on parallelizing PCG from various angles. SimPL [17], a quadratic placer for application-specific integrated circuits (ASICs), achieved 1.89X speedup for PCG by solving x- and y-directed wirelength simultaneously with 2 threads. Further speedup is possible by parallelizing matrix-vector operations in PCG, however, [18] showed that this technique cannot even achieve 2X speedup by using 16 threads in their experiments.

For nonlinear placers, wirelength and other constraints (e.g. cell density) are modeled into one single nonlinear objective, optimizing which becomes the runtime bottleneck. Unlike quadratic placers, the x and y directions in nonlinear placers cannot be decomposed into two independent components, therefore, parallelizing nonlinear placers is even harder.

A recent work, POLAR 3.0 [18], presented a quadratic placement parallelization framework for ASICs based on geometric partitioning. In particular, they divided cells into partitions based on their physical locations, then performed PCG and rough legalization [17] for each partition separately in parallel. In order to reduce solution quality loss, full-netlist placement was performed periodically to allow inter-partition cell moving. Although their approach achieved promising results (4X speedup with 1.2% wirelength degradation by using 16 threads) on ASIC benchmarks, similar performance gain might not be reproducible to FPGA designs for the following two reasons: 1) placeable cells and nets in FPGA designs typically have more pins compared with ASIC designs (as shown in Table. I), which leads to a stronger inter-partition connection and a larger quality degradation and 2) various FPGA architecture constraints (e.g. clock legalization rules [19]) impose difficulties to the parallelization of rough legalization.

TABLE I: Comparison of Recent FPGA and ASIC Placement Contest Benchmarks

Benchmark Suite		Avg. Num. Pins Per Mov. Cell	Avg. Num. Mov. Pins Per Net
FPGA	ISPD'16	4.99	4.95
	ISPD'17	4.16	4.15
ASIC	ICCAD'14	3.07	3.06
	ICCAD'15	3.13	2.99

In this paper, we propose a highly parallelized quadratic placement framework for FPGAs, UTPlaceF 3.0, which takes full advantages of modern multi-core CPUs and delivers an appealing performance boost. Besides, the placement quality loss also is taken care of with only very little runtime overhead. The major contributions of this work are highlighted as follows.

- We propose a placement-driven block-Jacobi preconditioning technique that transforms a placement problem into a set of independent sub-problems, which can be solved in parallel.
- We propose a parallelized incremental placement correction technique to reduce placement quality degradation.

- ISPD'16 benchmark suite demonstrates the effectiveness of our framework. On average, UTPlaceF 3.0 achieves 5X speedup by using 16 threads with only 3.0% wirelength degradation.

The rest of this paper is organized as follows. Section II reviews the preliminaries of quadratic placement. Section III systematically studies the performance bottlenecks of state-of-the-art quadratic placers. Section IV gives the details of UTPlaceF 3.0 algorithms. Section V shows the experimental results, followed by the conclusion and future work in Section VI.

II. Preliminaries

A. Quadratic Placement

An FPGA netlist can be represented as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$ is the set of cells, and $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ is the set of nets. Let $\mathbf{x} = \{x_1, x_2, ..., x_{|\mathcal{V}|}\}$ and $\mathbf{y} = \{y_1, y_2, ..., y_{|\mathcal{V}|}\}$ be the x and y coordinates of all cells. The wirelength-driven global placement problem is to determine cell position vectors $\mathbf{x}$ and $\mathbf{y}$ such that the total wirelength is minimized. Wirelength is measured by half-perimeter wirelength (HPWL) defined as follows.

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \sum_{e \in \mathcal{E}} \{\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j|\}. \quad (1)$$

As HPWL is not differentiable everywhere, quadratic placers approximate it by squared Euclidean distance between cells using various net models, such as hybrid net model [20] and bound-to-bound (B2B) net model [21]. Therefore, the wirelength cost function in quadratic placers is defined as

$$W(\mathbf{x}, \mathbf{y}) = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_x^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_y^T \mathbf{y} + const. \quad (2)$$

Since both Hessian matrices Q_x and Q_y in Eq. (2) are SPD, minimizing $W(\mathbf{x}, \mathbf{y})$ is equivalent to solving the following two linear systems.

$$Q_x \mathbf{x} = -\mathbf{c}_x, \quad (3a)$$
$$Q_y \mathbf{y} = -\mathbf{c}_y. \quad (3b)$$

To eliminate cell overlapping, most state-of-the-art quadratic placers [14]–[16] adopted a cell-spreading technique, called rough legalization [17], which adds extra spreading force pointing to cells' anchor locations (i.e., locations that satisfy cell density constraint). After each rough legalization execution, linear systems (3a) and (3b) are updated accordingly and solved again. The loop of solving linear systems and performing rough legalization is repeated until cells are fully spread out.

III. Empirical Runtime Study

In this section, we will empirically analyze the runtime bottleneck of a state-of-the-art quadratic placer and evidence that achieving a highly scalable parallel placement is indeed a challenging problem.

A representative rough legalization-based quadratic placement flow is presented in Fig. 1. To show its runtime bottlenecks, we implemented a pure wirelength-driven UTPlaceF[1] [14] and performed experiments on ISPD'16 benchmark suites with single thread. As illustrated in Fig. 2, the three major runtime contributors, on average, are solving linear systems (85.1%), rough legalization (7.5%), and linear system construction (7.3%), respectively.

[1]We removed routability optimization and global-move refinement in the original UTPlaceF.

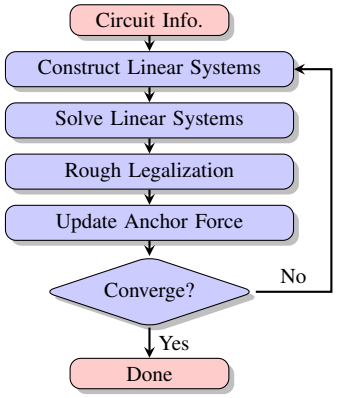

Fig. 1: A representative rough legalization-based quadratic placement flow.

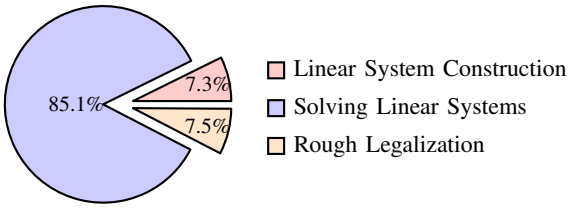

Fig. 2: Normalized runtime of the three major runtime contributors in our pure wirelength-driven UTPlaceF on ISPD'16 benchmark suite.

Due to the decomposability of x- and y-directed wirelength in quadratic placement, the linear systems (3a) and (3b) can be constructed and solved perfectly in parallel using two threads. However, further parallelization becomes harder and inefficient.

The most efficient linear system construction approach is to scan and fill Hessian matrices Q_x and Q_y in net-by-net manner. Considering multiple nets might contribute to the same entry in Q_x and Q_y (e.g. multiple nets share the same cell), there are two potential parallelization strategies: 1) processing nets in parallel and adding mutex lock[2] to each non-zero entry and 2) partitioning nets into groups and constructing partial Hessian matrices for each group in parallel, then, joining all partial Hessian matrices together. As can be seen, both strategies introduce considerable runtime overhead, therefore, linear system construction is difficult to be parallelized.

Solving linear systems, the most time-consuming step, is inherently hard to be parallelized as well due to the iterative nature of PCG. Experiments in [18] showed that only 1.8X speedup can be achieved by using the cutting-edge PCG solver in Intel MKL library [22] on an 8-core machine.

Unless special care is taken, rough legalization can only handle one cell density hotspots at a time, since spreading windows of multiple hotspots might intersect to each other. For each hotspot, the frequent cell sortings for horizontal and vertical cell spreading dominates the runtime. Although parallelism can be applied here, experiments in [17] only achieved 1.62X speedup by using 8 threads.

To manifest the placement parallelization crisis, we further parallelized our pure wirelength-driven UTPlaceF in the following way utilizing OpenMP [23]: 1) constructing and solving linear systems for x and y in parallel, 2) enabling parallelization for matrix-vector operations in PCG if more than two threads are available, and 3) substituting sequential sortings in rough legalization with their

[2]Mutex (mutually exclusive) lock is a mechanism that guarantees a resource can only be accessed by one thread at a time in a multi-threading environment.

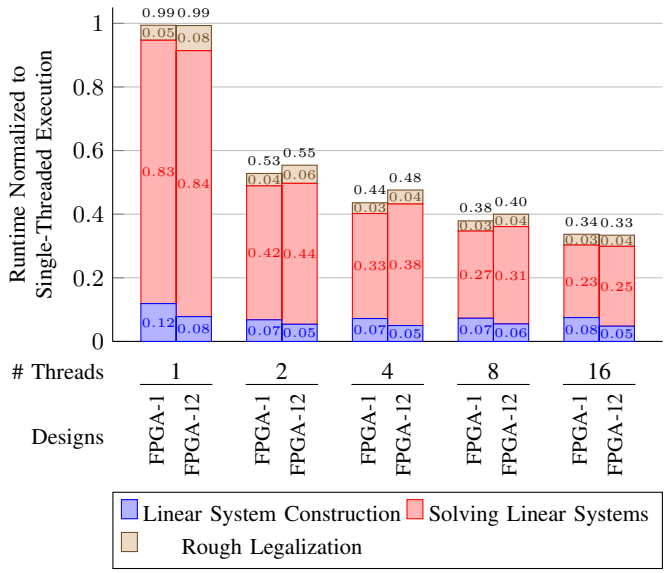

Fig. 3: Runtime scaling of the three major runtime contributors in our pure wirelength-driven UTPlaceF by multi-threading. FPGA-1 and FPGA-12 contain 0.1M and 1.1M cells, respectively.

equivalent parallel version in GNU libstdc++ parallel mode [24]. We then performed experiments using a 2.60 GHz Intel Xeon E5-2690 v3 CPU with 24 cores on two representative (the smallest and the largest) designs in ISPD'16 benchmark suite. 1, 2, 4, 8, and 16 threads were enabled respectively. Fig. 3 presents the runtime scaling of the three major runtime contributors. As shown, most PCG speedup is from simultaneously solving x and y (1 thread vs 2 threads) and it plateaus out quickly on matrix-vector level parallelization (from 2 threads to 16 threads). Both linear system construction and rough legalization scale poorly. The overall speedup saturates at around 3X.

In summary, decomposing x and y and low-level parallelization (i.e., sortings and matrix-vector operations) alone can only achieve about 3X speedup. In this work, we will explore innovative high-level approaches to break this parallelism wall.

IV. UTPlaceF 3.0 Algorithms

A. Overall Flow

The overall flow of UTPlaceF 3.0 is illustrated in Fig. 4. The whole flow starts with the initial placement consisting of one pass of linear system construction and solving followed by rough legalization and anchor force updating. After that, the netlist is divided into several sub-netlists utilizing hypergraph min-cut partitioning techniques. It should be noted that the resulting sub-netlists are not necessarily physically separated. In practice, cells from different sub-netlists are most likely mixed together. Once the partitioning is done, the linear systems of each sub-netlist are constructed and solved independently in parallel. However, because of the interdependency among sub-netlists (i.e., inter-partition nets), sub-netlist placement might not converge to the optimal solution, which leads to placement quality loss. Therefore, an incremental placement correction step, which takes the inter-partition dependency into consideration, is performed to reduce the quality degradation after all sub-netlist placements are done. Finally, rough legalization and anchor force updating are executed. The loop of parallelized placement, rough legalization, and anchor force updating is repeated until the wirelength converges.

B. Placement-Driven Block-Jacobi Preconditioning

As demonstrated by the empirical study in Section III, PCG dominates the total runtime of global placement. Thus, parallelizing

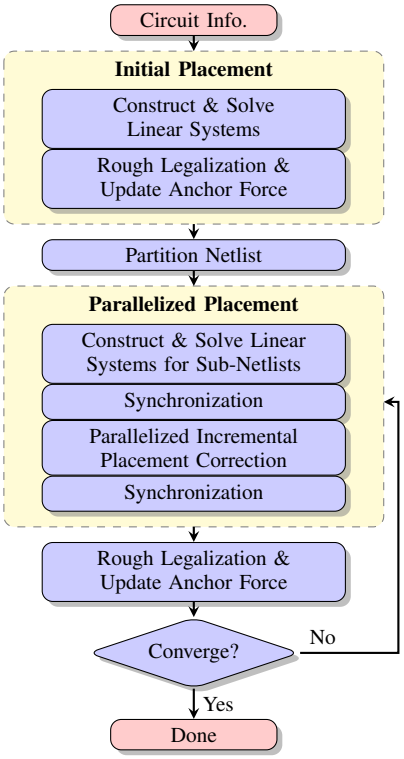

Fig. 4: The overall flow of UTPlaceF 3.0.

PCG is of top priority. An effective PCG parallelization scheme is to leverage the strength of block-Jacobi method. The block-Jacobi method essentially exploits the divide-and-conquer approach to transform a large linear system into a set of independent smaller sub-systems and solve each of them individually.

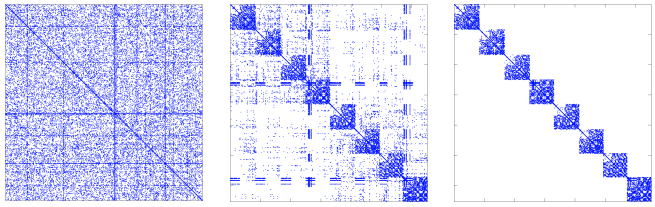

Fig. 5: An example of building a block-Jacobi preconditioner from a sparse SPD matrix Q based on an 8-way partitioning. (a) The matrix Q. (b) After an 8-way partitioning by row/column permutation. (c) The resulting block-Jacobi preconditioner of Q.

Fig. 5 shows an example of applying an 8-way block-Jacobi method on a sparse SPD matrix Q. The first step (from Fig. 5(a) to Fig. 5(b)) is to find a good row/column permutation such that the sum of off-block-diagonal non-zero entries are minimized. As an SPD matrix can be represented as an undirected weighted graph (UWG), finding the best permutation for k diagonal blocks is equivalent to finding the k-way min-cut partitioning of the induced UWG. It should be noted that the permutated Q (Fig. 5b) is mathematically equivalent to the original Q (Fig. 5a), thus, with proper permutation on decision vectors ($\mathbf{x}$ and $\mathbf{y}$) and right-hand sides ($\mathbf{c}_x$ and $\mathbf{c}_y$), a permutated linear system is also equivalent to the original one. After the permutation, the block-Jacobi preconditioner in Fig. 5(c) can be obtained by simply ignoring all off-block-diagonal non-zero entries in the permutated matrix in Fig. 5(b).

From placement point of view, each diagonal block in the block-Jacobi preconditioner corresponds to a partition (sub-netlist) and solving linear systems (3a) and (3b) with block-Jacobi preconditioner is physically equivalent to performing placement for each partition individually by assuming all off-partition cells are fixed at location $(x = 0, y = 0)$[3]. Although this approach scales almost linearly with the number of partitions created, the placement quality would degrade dramatically as wrong physical locations are assumed for cells in inter-partition nets.

To mitigate this problem, rather than using (0, 0) as the fixed point, off-partition cells are considered fixed at their locations induced from the linear system solutions of the previous placement iteration. An example of a three-pin net spanning three partitions is shown in Fig. 6. We call this process "placement-driven block-Jacobi preconditioning".

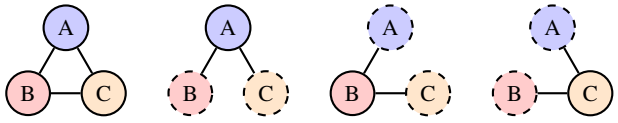

Fig. 6: An inter-partition net spanning three partitions. (a) illustrates the net in the original netlist. (b), (c), and (d) show the net in the partition containing A, B, and C, respectively. Dashed circles represent off-partition cells that are fixed at their last locations.

Given a netlist consisting of the set of cell $\mathcal{V}$, a partitioning $\mathcal{P}$ on $\mathcal{V}$, and the $\mathcal{P}$-permutated linear systems (3a) and (3b), if we denote the belonging partition of cell $i \in \mathcal{V}$ by $\pi(i)$ and denote the placement-driven block-Jacobi preconditioned linear systems of (3a) and (3b) by

$$\widehat{Q}_x \mathbf{x} = -\widehat{\mathbf{c}}_x, \tag{4a}$$

$$\widehat{Q}_y \mathbf{y} = -\widehat{\mathbf{c}}_y, \tag{4b}$$

then, $\widehat{Q}$ ($\widehat{Q}_x$ and $\widehat{Q}_y$) and $\widehat{\mathbf{c}}$ ($\widehat{\mathbf{c}}_x$ and $\widehat{\mathbf{c}}_y$) can be defined as

$$\widehat{Q}_{i,j} = \begin{cases} 0, & \text{if } \pi(i) \neq \pi(j), \forall i, j \in \mathcal{V}, \\ Q_{i,j}, & \text{otherwise}, \end{cases} \tag{5a}$$

$$\widehat{\mathbf{c}}_i = \mathbf{c}_i + \sum_{k \in \mathcal{V} \setminus \pi(i)} Q_{i,k} \cdot l_k, \forall i \in \mathcal{V}, \tag{5b}$$

where l_k denotes the (x or y) coordinate of cell $k \in \mathcal{V}$ in the previous placement iteration. Compared with normal block-Jacobi preconditioning, our placement-driven block-Jacobi preconditioning only differs by the right-hand sides in linear systems. Therefore, the SPD property is retained in linear systems (4a) and (4b) and diagonal blocks can still be solved independently by PCG.

Since the matrices Q_x and Q_y are placement dependent, from quality wise, it is ideal to perform two (one each for x and y) UWG min-cut partitionings for each placement iteration. However, doing so will result in a dramatic amount of runtime overhead. Therefore, only one partitioning is done right before the first parallelized placement step as shown in Fig. 4. Considering Q_x and Q_y change after every placement iteration, hypergraph min-cut partitioning is adopted to approximate the varied optimal matrix partitionings.

[3]The wirelength cost in x direction for two cells can be written as $w_{i,j}(x_i - x_j)^2$. If cell i and j are not in the same partition, the term $-2w_{i,j}x_i x_j$ would be ignored in the block-Jacobi preconditioner. Therefore, this cost term becomes $w_{i,j}(x_i - 0)^2 + w_{i,j}(x_j - 0)^2$. For cell i (j), this is equivalent to assuming cell j (i) is fixed at $x = 0$. Similar conclusion is hold for y direction.

C. Parallelized Incremental Placement Correction (PIPC)

Although applying placement-driven block-Jacobi preconditioning can reduce quality loss to some extent, it is no longer effective enough once more partitions (e.g. more than 4) are created. To further mitigate placement quality degradation introduced by partitioning, the idea of additive correction multigrid method [25] is adopted to incrementally correct the solutions of linear systems (4a) and (4b).

Without loss of generality, we only discuss x direction in the rest of this session, but conclusions that hold for x are also applicable to y.

Let $\mathbf{x}^{(0)}$ be the solution of linear system (4a), that is

$$\widehat{Q}\mathbf{x}^{(0)} = -\widehat{\mathbf{c}}, \tag{6}$$

where $\widehat{Q}$ and $\widehat{\mathbf{c}}$ are defined in Eq. (5a) and Eq. (5b), respectively. If we can find $\Delta\mathbf{x}$ satisfying

$$Q\Delta\mathbf{x} = -\mathbf{c} - Q\mathbf{x}^{(0)}, \tag{7}$$

the solution of the original linear system (3a) will be $\mathbf{x}^{(0)} + \Delta\mathbf{x}$. Intuitively, $\Delta\mathbf{x}$ represents the discrepancy between current placement $\mathbf{x}^{(0)}$ and the optimal placement $\mathbf{x}^\star$ and it is, physically, a very local and smooth perturbation around the placement $\mathbf{x}^{(0)}$.

Now we present the approach to find $\Delta\mathbf{x}$. Let $\mathbf{r}^{(0)} = -\mathbf{c} - Q\mathbf{x}^{(0)}$ and $N = \widehat{Q} - Q$. Then, solving linear system (7) is equivalent to solving

$$\widehat{Q}\Delta\mathbf{x} = N\Delta\mathbf{x} + \mathbf{r}^{(0)}. \tag{8}$$

The iterative method now becomes solving $\Delta\mathbf{x}^{(k)}$ for $k \geq 1$ with given $\Delta\mathbf{x}^{(0)}$ by

$$\widehat{Q}\Delta\mathbf{x}^{(k+1)} = N\Delta\mathbf{x}^{(k)} + \mathbf{r}^{(0)}, \tag{9}$$

which can be further written as

$$\Delta\mathbf{x}^{(k+1)} = \Delta\mathbf{x}^{(k)} + \widehat{Q}^{-1}(\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}). \tag{10}$$

Intuitively, if $\Delta\mathbf{x}^{(k)}$ and $\Delta\mathbf{x}^{(k+1)}$ are becoming closer and closer (i.e., $\Delta\mathbf{x}^{(k)}$ converges), we have $\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}$ approaches to 0. This indicates that once $\Delta\mathbf{x}^{(k)}$ converges, it will always converge to the same solution. To formally prove the convergence, consider the following lemma [26].

Lemma 1. *Let $Q = \widehat{Q} - N$, with Q and $\widehat{Q}$ symmetric and positive definite. If the matrix $2\widehat{Q} - Q$ is positive definite, then the iterative method defined in Eq.* (10) *is convergent for any choice of the initial $\Delta\mathbf{x}^{(0)}$. Moreover, the convergence of the iteration is monotone with respect to the norm $||\cdot||_Q$ (i.e., $||\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k+1)}|| < ||\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}||$, $k = 0, 1, ...$).*

Now we can conclude the following proposition.

Proposition 2. *By picking any placement-driven block-Jacobi preconditioner of Q as $\widehat{Q}$, the iterative method defined in Eq.* (10) *is monotonically convergent for any choice of $\Delta\mathbf{x}^{(0)}$.*

Proof. Since $|Q_{i,i}| > \sum_{j \neq i} |Q_{i,j}|, \forall i \in \mathcal{V}$, both Q and $\widehat{Q}$ are symmetric strictly diagonally dominant matrices. By the definition of $\widehat{Q}$ in Eq. (5a), the matrix $\Omega = 2\widehat{Q} - Q$ can be defined as

$$\Omega_{i,j} = \begin{cases} -Q_{i,j}, & \text{if } \pi(i) \neq \pi(j), \forall i, j \in \mathcal{V}, \\ Q_{i,j}, & \text{otherwise}. \end{cases}$$

So $\Omega = 2\widehat{Q} - Q$ is also a symmetric strictly diagonally dominant matrix. A symmetric strictly diagonally dominant real matrix with nonnegative diagonal entries is positive definite. Therefore, $2\widehat{Q} - Q$ is positive definite and, thus, the iterative scheme defined by Eq. (10) will converge monotonically. □

Proposition 2 reveals two important properties of the iterative method in Eq. (10): 1) with sufficient iterations, the exact solution, $\Delta\mathbf{x}$, of linear system (7) can be reached, hence, we can obtain the exact solution, $\mathbf{x}^{(0)} + \Delta\mathbf{x}$, of linear system (3a) and 2) since the convergence is monotone, more iterations guarantees better solutions (i.e., solutions that are closer to the exact solution). These two properties imply that, by using this iterative method, placement quality and runtime can be perfectly traded to each other. Thus, different trade-offs can be made accordingly under different scenarios.

Another attractive property of the iterative method in Eq. (10) is its strong parallelizability. Almost all the runtime of solving Eq. (10) is taken by computing $\widehat{Q}^{-1}(\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)})$, which is equivalent to solving

$$\widehat{Q}\mathbf{x} = \mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}. \tag{11}$$

Recall that $\widehat{Q}$ is the placement-driven block-Jacobi preconditioner

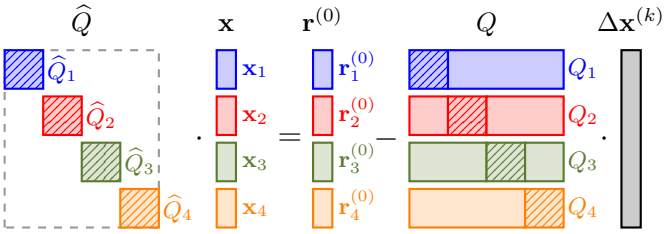

Fig. 7: Parallelization scheme of solving linear system (11) with four partitions. Shaded regions represent diagonal blocks in $\widehat{Q}$ and Q. Different colors denote different partitions that can be solved in parallel.

of Q defined in Eq. (5a). Fig. 7 illustrates the parallelization scheme of solving this linear system by an example with four partitions. By blocking matrix Q along rows, each strap in Q can be constructed independently and solving linear system (11) becomes solving the following sub-system for each partition separately and then assembling their solutions ($\mathbf{x}_i$) together.

$$\widehat{Q}_i\mathbf{x}_i = \mathbf{r}_i^{(0)} - Q_i\Delta\mathbf{x}^{(k)}, \forall i \in \mathcal{P}, \tag{12}$$

where $\widehat{Q}_i$, x_i, $\mathbf{r}_i^{(0)}$, and Q_i are partial matrices and vectors corresponding to partition i as illustrated in Fig. 7, and $\mathcal{P}$ denotes the set of partitions.

Algorithm 1 summarizes the overall flow of our parallelized incremental placement correction (PIPC) scheme. The partial matrix Q_i of Q for each partition is constructed in parallel from line 1 to line 3. Then the linear system (12) for each partition is solved in parallel from line 7 to line 9. After that, the solution of linear system (11) is obtained by assembling partial solutions together in line 10. The solution after each correction iteration is incrementally updated in line 11. The loop from line 6 to line 13 is repeated until the correction iteration limit I is reached.

D. Varied PIPC Configuration

In general, placement quality degrades as the number of partitions increases. So it is not wise to perform the same amount of PIPC for different number of partitions, which might be overkill for small partition counts but insufficient for a large number of partitions. To resolve this issue, PIPC is configured based on the number of partitions created accordingly.

In UTPlaceF 3.0, we configure three related variables: 1) the frequency of applying PIPC, 2) the number of correction iterations in each PIPC, and 3) the number of PCG iterations in PIPC. For small partition counts, good placement quality can still be maintained by infrequent PIPC (e.g. once every 3 placement iterations) with only

Algorithm 1 Parallelized Incremental Placement Correction

Input: The set of partitions $\mathcal{P}$, the solution of linear system (4a) $\mathbf{x}^{(0)}$, and the number of correction iterations I.
Output: A better solution than $\mathbf{x}^{(0)}$.

1: **parallel for** each $i \in \mathcal{P}$ **do**
2: Construct Q_i; ▷ See Fig. (7)
3: **end parallel for**
4: $k \leftarrow 0$;
5: $\Delta\mathbf{x}^{(0)} \leftarrow 0$;
6: **while** $k < I$ **do**
7: **parallel for** each $i \in \mathcal{P}$ **do**
8: Solve $\widehat{Q}_i\Delta\mathbf{x}_i^{(k+1)} = \mathbf{r}_i^{(0)} - Q_i\Delta\mathbf{x}^{(k)}$;
9: **end parallel for**
10: $\Delta\mathbf{x}^{(k+1)} \leftarrow (\Delta\mathbf{x}_1^{(k+1)}, \Delta\mathbf{x}_2^{(k+1)}, ..., \Delta\mathbf{x}_{|\mathcal{P}|}^{(k+1)})$;
11: $\Delta\mathbf{x}^{(k+1)} \leftarrow \Delta\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k+1)}$;
12: $k \leftarrow k + 1$;
13: **end while**
14: **return** $\mathbf{x}^{(0)} + \Delta\mathbf{x}^{(I)}$;

a few correction iterations (e.g. 1 or 2) each time. As the partition count increases, PIPC frequency and correction iterations need to be increased properly to control quality degradation. The third variable, PCG iteration count in PIPC, is tuned to further save runtime. Since the solution of PIPC ($\Delta\mathbf{x}$) is essentially a local perturbation of the existing placement ($\mathbf{x}^{(0)}$), its magnitude is typically much smaller than $\mathbf{x}^{(0)}$. Therefore, we can save some PCG iterations in PIPC without losing too much placement quality. The detailed configuration will be further discussed in Section V-A.

V. Experimental Results

UTPlaceF 3.0 is implemented in C++ and compiled by g++ 4.8.4. OpenMP 4.0 [23] is used to support multi-threading, GNU libstdc++ parallel mode [24] is used to parallelize critical sortings in rough legalization, hypergraph partitioning tool PaToh [27] is used to partition netlists, and the PCG in Eigen3 [28] is used to solve sparse SPD linear systems. All the experiments are performed on a Linux machine running with Intel Xeon E5-2690 v3 CPUs (2.60 GHz, 24 cores, and 30M L3 cache) and 64 GB RAM.

TABLE II: ISPD'16 Placement Contest Benchmarks Statistics

Benchmark	#LUT	#FF	#RAM	#DSP
FPGA-1	50K	55K	0	0
FPGA-2	100K	66K	100	100
FPGA-3	250K	170K	600	500
FPGA-4	250K	172K	600	500
FPGA-5	250K	174K	600	500
FPGA-6	350K	352K	1000	600
FPGA-7	350K	355K	1000	600
FPGA-8	500K	216K	600	500
FPGA-9	500K	366K	1000	600
FPGA-10	350K	600K	1000	600
FPGA-11	480K	363K	1000	400
FPGA-12	500K	602K	600	500
Resources	538K	1075K	1728	768

The benchmark suite released by Xilinx for ISPD'16 FPGA placement contest [29] is used to evaluate the efficiency of UTPlaceF 3.0. The statistics of the benchmarks are listed in Table II. Since this work is focused on placement parallelization, all routability optimizations are discarded and only wirelength is considered. In the experiments, the CLB resource demands of all LUTs and FFs are set to 0.08 and the target density is set to 1.0 for all benchmarks. In the target FPGA architecture, considering a vertical route needs to go through about twice as many switch boxes as a horizontal route needs for the same

length, the wirelength is measured by the scaled HPWL (sHPWL) defined as

$$sHPWL = 0.5 \cdot HPWL_x + HPWL_y, \tag{13}$$

where $HPWL_x$ and $HPWL_y$ denote the x- and y-directed components of HPWL in Eq. (1), respectively.

A. Parallelization Configuration

TABLE III: Parallelization Configuration of UTPlaceF 3.0 Under Different Number of Threads

# Threads	1	2	4	8	16
# Partitions	1	1	2	4	8
# PCG Iter.	250	250	250	250	250
PIPC Period	-	-	-	3	1
# Iter. of each PIPC	-	-	-	1	1
# PCG Iter. in PIPC	-	-	-	150	50

The parallelization configuration used for our experiments is presented in Table III. If N ($N > 1$) threads are available, we always generate $\lfloor \frac{N}{2} \rfloor$ partitions and deal x and y separately using 2 threads in each partition. The number of PCG iterations for solving linear systems (3a), (3b), (4a), and (4b) are universally set to 250. PIPC is disabled for 1, 2, and 4 threads, since good placement quality can be achieved with placement-driven block-Jacobi preconditioning alone. For the case of 8 threads, PIPC is applied every 3 placement iterations and each time only one correction iteration is performed. For 16 threads, PIPC needs to be applied at every placement iteration to maintain good solution quality. The numbers of PCG iterations in PIPC are set to 150 and 50, respectively, for 8 and 16 threads. Although fewer PCG iterations are performed for 16 threads, the quality can be guaranteed by its more frequent PIPC calls.

B. PIPC Effectiveness Validation

The experimental results without PIPC (but with placement-driven block-Jacobi preconditioning) are presented in Table IV. We report the sHPWL (WL) in the unit of 10^3, runtime (RT) in the unit of second, and their ratios (WLR and RTR) compared to corresponding 1-thread execution.

On average, without PIPC, UTPlaceF 3.0 achieves 1.7X, 2.8X, 4.4X, and 5.9X speedup by using 2, 4, 8, and 16 threads. With 8 and 16 threads, the wirelength degrades by 2.4% and 6.5%, respectively.

Since PIPC is not applied for the cases of 1, 2, and 4 threads as shown in Table III, we only report the results with PIPC enabled for 8 and 16 threads in Table V. As can be seen, PIPC effectively reduces the wirelength degradation from 2.4% and 6.5% to 1.7% and 3.0% with only a little runtime overhead. With PIPC, UTPlaceF 3.0 can still achieve 5X speedup by using 16 threads.

It is worthwhile to mention that PIPC is a general technique that can be configured for different runtime and quality trade-offs. The results shown in Table V is only a set of experiments to demonstrate the effectiveness of PIPC by using the configuration in Table III.

C. Runtime Analysis

Table VI presents the runtime breakdown of UTPlaceF 3.0 under different number of threads. We divide the total runtime into components of solving linear systems, linear system construction, rough legalization, netlists partitioning, PIPC, and others. The normalized runtime (Norm. RT) and runtime percentage (RT %) are reported for each component under each thread count. As the number of threads increases, the runtime of solving linear systems reduces nearly linearly. However, the speedup of linear system construction and rough legalization saturate quickly. For the case of 16 threads, the runtime taken by each component becomes pretty much comparable.

According to the runtime breakdown, we can see that further speedup with more threads becomes difficult for the following four main reasons: 1) solving linear systems does not dominate the total runtime anymore, thus, the overall gain from it becomes limited, 2) linear system construction and rough legalization scales poorly, 3) the runtime of netlist partitioning is almost linear to the number of partitions, and 4) more PIPC would be needed to maintain good placement quality. However, it is still possible to push the runtime down to the limit by the following several improvements: 1) combine low-level (matrix-vector operations) parallelization with our partition-based framework, 2) combine the high-level parallelization scheme proposed by [17] with our parallelization for critical sortings, and 3) use parallelized partitioning tools (the one in UTPlaceF 3.0 now is single-threaded).

VI. Conclusion and Future Work

In this paper, we have proposed a parallelization framework for modern FPGA global placement, UTPlaceF 3.0. A placement-driven block-Jacobi preconditioning technique as well as a parallelized incremental placement correction technique are proposed for boosting the overall performance of FPGA global placement. Our experiments demonstrate that, by fully leveraging today's multi-core systems, UTPlaceF 3.0 can achieve significant speedup while maintaining competitive placement quality.

In our future work, we plan to parallelize packing and detailed placement algorithms to further reduce the runtime of the whole FPGA placement flow.

Acknowledgment

This work is supported in part by Intel Corporation. The authors would like to thank Dr. Mahesh Iyer at Intel for helpful discussions.

References

[1] Xilinx Inc., http://www.xilinx.com, accessed: 2017-8-1.
[2] Intel (Altera) Corp., http://www.altera.com, accessed: 2017-8-1.
[3] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 1997, pp. 213–222.
[4] G. Chen and J. Cong, "Simultaneous timing driven clustering and placement for fpgas," *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pp. 158–167, 2004.
[5] ——, "Simultaneous placement with clustering and duplication," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 3, pp. 740–772, 2006.
[6] P. Maidee, C. Ababei, and K. Bazargan, "Fast timing-driven partitioning-based placement for island style FPGAs," in *ACM/IEEE Design Automation Conference (DAC)*, 2003, pp. 598–603.
[7] ——, "Timing-driven partitioning-based placement for island style FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 3, pp. 395–406, 2005.
[8] Y. Xu and M. A. Khalid, "QPF: efficient quadratic placement for FPGAs," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2005, pp. 555–558.
[9] P. Gopalakrishnan, X. Li, and L. Pileggi, "Architecture-aware FPGA placement using metric embedding," in *ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 460–465.
[10] M. Xu, G. Gréwal, and S. Areibi, "StarPlace: A new analytic method for FPGA placement," *Integration, the VLSI Journal*, vol. 44, no. 3, pp. 192–204, 2011.
[11] M. Gort and J. H. Anderson, "Analytical placement for heterogeneous FPGAs," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 143–150.
[12] T.-H. Lin, P. Banerjee, and Y.-W. Chang, "An efficient and effective analytical placer for FPGAs," in *ACM/IEEE Design Automation Conference (DAC)*, 2013, pp. 10:1–10:6.
[13] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang, "Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014, pp. 647–654.

TABLE IV: Comparison of UTPlaceF 3.0 without PIPC Under Different Number of Threads

Designs	1 Thread				2 Threads				4 Threads				8 Threads				16 Threads			
	WL	RT	WLR*	RTR†	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR	WL	RT	WLR	RTR
FPGA-1	227	44	1.000	1.000	227	24	1.000	0.546	225	15	0.987	0.341	233	10	1.025	0.220	233	7	1.027	0.165
FPGA-2	437	72	1.000	1.000	437	40	1.000	0.557	427	29	0.977	0.394	450	17	1.029	0.238	474	12	1.085	0.172
FPGA-3	1663	232	1.000	1.000	1663	131	1.000	0.567	1676	85	1.008	0.368	1750	51	1.052	0.222	1771	39	1.065	0.169
FPGA-4	3375	250	1.000	1.000	3375	148	1.000	0.589	3361	90	0.996	0.359	3408	61	1.010	0.242	3449	43	1.022	0.172
FPGA-5	6504	293	1.000	1.000	6504	173	1.000	0.589	6322	112	0.972	0.380	6489	71	0.998	0.243	6579	54	1.011	0.185
FPGA-6	3144	450	1.000	1.000	3144	249	1.000	0.555	3188	153	1.014	0.341	3106	103	0.988	0.229	3301	75	1.050	0.167
FPGA-7	5851	446	1.000	1.000	5851	251	1.000	0.562	5910	164	1.010	0.368	6004	99	1.026	0.223	6500	81	1.111	0.181
FPGA-8	5555	526	1.000	1.000	5555	298	1.000	0.567	5667	179	1.020	0.340	6059	119	1.091	0.225	6077	86	1.094	0.163
FPGA-9	7318	580	1.000	1.000	7318	323	1.000	0.557	7551	206	1.032	0.355	7564	136	1.034	0.235	7523	101	1.028	0.174
FPGA-10	3062	579	1.000	1.000	3062	316	1.000	0.546	3059	200	0.999	0.346	3058	128	0.999	0.221	3130	96	1.022	0.165
FPGA-11	6975	601	1.000	1.000	6975	335	1.000	0.558	7004	202	1.004	0.336	7053	133	1.011	0.221	8807	96	1.263	0.160
FPGA-12	3837	824	1.000	1.000	3837	407	1.000	0.494	3723	260	0.970	0.316	3943	168	1.028	0.205	3926	125	1.023	0.151
Geo. Mean	-	-	1.000	1.000	-	-	1.000	0.557	-	-	0.999	0.353	-	-	1.024	0.227	-	-	1.065	0.169

* WLR: Wirelength ratio compared to 1-thread execution.
† RTR: Runtime ratio compared to 1-thread execution.

TABLE V: Comparison of UTPlaceF 3.0 with PIPC Under Different Number of Threads

Designs	8 Threads				16 Threads			
	WL	RT	WLR	RTR	WL	RT	WLR	RTR
FPGA-1	230	11	1.013	0.256	232	9	1.020	0.200
FPGA-2	448	19	1.024	0.261	453	15	1.037	0.207
FPGA-3	1719	57	1.034	0.247	1674	46	1.007	0.201
FPGA-4	3406	68	1.009	0.270	3379	51	1.001	0.204
FPGA-5	6413	85	0.986	0.290	6549	64	1.007	0.217
FPGA-6	3125	114	0.994	0.254	3203	91	1.019	0.203
FPGA-7	6160	115	1.053	0.258	6507	90	1.112	0.201
FPGA-8	5848	130	1.053	0.247	6151	103	1.107	0.197
FPGA-9	7576	156	1.035	0.268	7341	114	1.003	0.197
FPGA-10	3043	150	0.994	0.258	3036	116	0.992	0.200
FPGA-11	7077	154	1.015	0.256	7287	113	1.045	0.189
FPGA-12	3845	187	1.002	0.227	3897	144	1.016	0.175
Geo. Mean	-	-	1.017	0.257	-	-	1.030	0.199

TABLE VI: Runtime Breakdown of UTPlaceF 3.0 Under Different Number of Threads

Components	1 Threads		2 Threads		4 Threads		8 Threads		16 Threads	
	Norm. RT*	RT %†	Norm. RT	RT %	Norm. RT	RT %	Norm. RT	RT %	Norm. RT	RT %
Solve Linear System	0.851	85.1%	0.449	80.6%	0.231	65.6%	0.125	48.5%	0.076	38.0%
Constr. Linear System	0.073	7.3%	0.047	8.4%	0.047	13.3%	0.034	13.4%	0.025	12.6%
Rough Legalization	0.075	7.5%	0.053	9.5%	0.042	11.9%	0.036	14.0%	0.033	16.7%
Partition Netlists	-	-	-	-	0.005	1.5%	0.009	3.6%	0.016	7.9%
PIPC	-	-	-	-	-	-	0.033	12.8%	0.034	16.9%
Others	0.001	0.1%	0.008	1.5%	0.027	7.7%	0.020	7.7%	0.016	7.9%
Total	1.000	100.0%	0.557	100.0%	0.353	100.0%	0.257	100.0%	0.199	100.0%

* Norm. RT: Normalized runtime of each component compared to the total runtime of 1-thread execution.
† RT %: The percentage of total runtime taken by each component under different threads.

[14] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 66:1–66:7.

[15] 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 *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 67:1–67:8.

[16] R. Pattison, Z. Abuowaimer, S. Areibi, G. Grewal, and A. Vannelli, "GPlace: A congestion-aware placement tool for ultrascale FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 68:1–68:7.

[17] 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 (TCAD)*, vol. 31, no. 1, pp. 50–60, 2012.

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

[19] ISPD 2017 Clock-Aware FPGA Placement Contest, http://www.ispd.cc/contests/17/, accessed: 2017-8-1.

[20] N. Viswanathan and C. C. Chu, "FastPlace: efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 5, pp. 722–733, 2005.

[21] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Kraftwerk2-A fast force-directed quadratic placement approach using an accurate net model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 8, pp. 1398–1411, 2008.

[22] Intel Math Kernel Library, https://software.intel.com/en-us/mkl, accessed: 2017-8-1.

[23] OpenMP 4.0, http://www.openmp.org/, accessed: 2017-8-1.

[24] GNU libstdc++ Parallel Mode, https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html, accessed: 2017-8-1.

[25] B. R. Hutchinson and G. D. Raithby, "A multigrid method based on the additive correction strategy," *Numerical Heat Transfer, Part A: Applications*, vol. 9, no. 5, pp. 511–537, 1986.

[26] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical mathematics*. Springer Science & Business Media, 2010.

[27] PaToH, http://bmi.osu.edu/umit/software.html, accessed: 2017-8-1.

[28] G. Guennebaud, B. Jacob *et al.*, "Eigen3," http://eigen.tuxfamily.org.

[29] ISPD 2016 Routability-Driven FPGA Placement Contest, http://www.ispd.cc/contests/16/ispd2016_contest.html, accessed: 2017-8-1.