UTPlaceF 3.0: A Parallelization Framework for Modern FPGA Global Placement

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-enhancing 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>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA ISPD'16</td>
<td>4.99</td>
<td>4.95</td>
</tr>
<tr>
<td>ISPD'17</td>
<td>4.16</td>
<td>4.15</td>
</tr>
<tr>
<td>ASIC ICCAD'14</td>
<td>3.07</td>
<td>3.06</td>
</tr>
<tr>
<td>ICCAD'15</td>
<td>3.13</td>
<td>2.99</td>
</tr>
</tbody>
</table>

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

\[
\text{HPWL}(x, y) = \sum_{e \in E} \left( \max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j| \right).
\] (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

\[
W(x, y) = \frac{1}{2} x^T Q_x x + c_x^T x + \frac{1}{2} y^T Q_y y + c_y^T y + \text{const.}
\] (2)

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

\[
Q_x x = -c_x,
\] (3a)
\[
Q_y y = -c_y.
\] (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 [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.

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. 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.
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 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 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 permuted Q (Fig. 5b) is mathematically equivalent to the original Q (Fig. 5a), thus, with proper permutation on decision vectors (x and y) and right-hand sides (c_x and c_y), a permuted 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 permuted 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 \((x = 0, y = 0)\). 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.](image)

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

\[
\hat{Q}_x x = -\hat{c}_x, \quad \hat{Q}_y y = -\hat{c}_y, \tag{4a}
\]

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

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

\[
\hat{c}_i = c_i + \sum_{k \in V \setminus \{i\}} Q_{i,k} \cdot l_k, \forall i \in V, \tag{5b}
\]

where \(l_k\) denotes the (x or y) coordinate of cell \(k \in V\) 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.

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

\[
\hat{Q} x^{(0)} = -\hat{c}, \tag{6}
\]

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

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

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

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

\[
\hat{Q} \Delta x = N \Delta x + r^{(0)}. \tag{8}
\]

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

\[
\hat{Q} \Delta x^{(k+1)} = N \Delta x^{(k)} + r^{(0)}, \tag{9}
\]

which can be further written as

\[
\Delta x^{(k+1)} = \Delta x^{(k)} + (\hat{Q}^{-1} r^{(0)} - Q \Delta x^{(k)}). \tag{10}
\]

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

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

Now we can conclude the following proposition.

**Proposition 2.** By picking any placement-driven block-Jacobi preconditioner of \(Q\) as \(\hat{Q}\), the iterative method defined in Eq. (10) is monotonically convergent for any choice of \(\Delta x^{(0)}\).
Proposition 2 reveals two important properties of the iterative method in Eq. (10): 1) with sufficient iterations, the exact solution, \( \Delta x \), of linear system (7) can be reached, hence, we can obtain the exact solution, \( x^{(0)} + \Delta x \), 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 \( Q^{-1}(r^{(0)} - Q\Delta x^{(k)}) \), which is equivalent to solving
\[
\hat{Q}x = r^{(0)} - Q\Delta x^{(k)}.
\]

Recall that \( \hat{Q} \) is the placement-driven block-Jacobi preconditioner 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 (\( x_i \)) together.
\[
\hat{Q}_i x_i = r_i^{(0)} - Q_i \Delta x^{(k)}, \forall i \in P,
\]
where \( \hat{Q}_i \), \( x_i \), \( r_i^{(0)} \), and \( Q_i \), are partial matrices and vectors corresponding to partition \( i \) as illustrated in Fig. 7, and \( P \) 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 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 (\( \Delta x \)) is essentially a local perturbation of the existing placement (\( x^{(0)} \)), its magnitude is typically much smaller than \( x^{(0)} \). 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 sorteds 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#LUT</th>
<th>#FF</th>
<th>#RAM</th>
<th>#DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA-1</td>
<td>50K</td>
<td>35K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FPGA-2</td>
<td>100K</td>
<td>66K</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>FPGA-3</td>
<td>250K</td>
<td>170K</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>FPGA-4</td>
<td>250K</td>
<td>172K</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>FPGA-5</td>
<td>250K</td>
<td>174K</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>FPGA-6</td>
<td>350K</td>
<td>352K</td>
<td>1000</td>
<td>600</td>
</tr>
<tr>
<td>FPGA-7</td>
<td>350K</td>
<td>355K</td>
<td>1000</td>
<td>600</td>
</tr>
<tr>
<td>FPGA-8</td>
<td>500K</td>
<td>216K</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>FPGA-9</td>
<td>500K</td>
<td>366K</td>
<td>1000</td>
<td>600</td>
</tr>
<tr>
<td>FPGA-10</td>
<td>350K</td>
<td>600K</td>
<td>1000</td>
<td>600</td>
</tr>
<tr>
<td>FPGA-11</td>
<td>480K</td>
<td>363K</td>
<td>1000</td>
<td>400</td>
</tr>
<tr>
<td>FPGA-12</td>
<td>500K</td>
<td>602K</td>
<td>600</td>
<td>500</td>
</tr>
</tbody>
</table>

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

\[ sHPWL = 0.5 \cdot HPWL_x + HPWL_y, \]  

(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

<table>
<thead>
<tr>
<th># Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td># Partitions</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>PCG Iter.</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>PIPC Period</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td># Iter. of each PIPC</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PCG Iter. in PIPC</td>
<td>-</td>
<td>-</td>
<td>150</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Designs</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
<th>16 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WL RT WLR</td>
<td>WL RT WLR</td>
<td>WL RT WLR</td>
<td>WL RT WLR</td>
<td>WL RT WLR</td>
</tr>
<tr>
<td>FPGA-1</td>
<td>227 44 1.000 1.000</td>
<td>227 24 1.000 0.546</td>
<td>225 15 0.987 0.341</td>
<td>233 10 1.025 0.220</td>
<td>233 7 1.027 0.165</td>
</tr>
<tr>
<td>FPGA-2</td>
<td>437 72 1.000 1.000</td>
<td>437 40 1.000 0.557</td>
<td>427 29 0.977 0.394</td>
<td>450 17 1.029 0.238</td>
<td>474 12 1.085 0.172</td>
</tr>
<tr>
<td>FPGA-3</td>
<td>1663 232 1.000 1.000</td>
<td>1663 131 1.000 0.567</td>
<td>1676 85 1.008 0.368</td>
<td>1750 51 1.052 0.222</td>
<td>1771 39 1.065 0.169</td>
</tr>
<tr>
<td>FPGA-4</td>
<td>3375 250 1.000 1.000</td>
<td>3375 148 1.000 0.589</td>
<td>3361 90 0.996 0.359</td>
<td>3408 61 1.010 0.242</td>
<td>3449 43 1.022 0.172</td>
</tr>
<tr>
<td>FPGA-5</td>
<td>6504 293 1.000 1.000</td>
<td>6504 173 1.000 0.589</td>
<td>6322 112 0.972 0.380</td>
<td>6489 71 1.000 0.388</td>
<td>6597 54 1.011 0.185</td>
</tr>
<tr>
<td>FPGA-6</td>
<td>3144 450 1.000 1.000</td>
<td>3144 249 1.000 0.555</td>
<td>3188 153 1.014 0.341</td>
<td>3106 103 0.988 0.229</td>
<td>3301 75 1.050 0.167</td>
</tr>
<tr>
<td>FPGA-7</td>
<td>5851 446 1.000 1.000</td>
<td>5851 251 1.000 0.562</td>
<td>5910 164 1.010 0.368</td>
<td>6004 99 1.002 0.226</td>
<td>6500 81 1.111 0.181</td>
</tr>
<tr>
<td>FPGA-8</td>
<td>5555 526 1.000 1.000</td>
<td>5555 298 1.000 0.557</td>
<td>5667 170 1.020 0.340</td>
<td>6059 119 1.091 0.225</td>
<td>6707 86 1.094 0.163</td>
</tr>
<tr>
<td>FPGA-9</td>
<td>7318 380 1.000 1.000</td>
<td>7318 323 1.000 0.557</td>
<td>7551 206 1.032 0.355</td>
<td>7564 136 1.034 0.235</td>
<td>7523 101 1.028 0.174</td>
</tr>
<tr>
<td>FPGA-10</td>
<td>3062 579 1.000 1.000</td>
<td>3062 316 1.000 0.546</td>
<td>3059 200 0.999 0.346</td>
<td>3058 128 0.999 0.291</td>
<td>3130 96 1.022 0.165</td>
</tr>
<tr>
<td>FPGA-11</td>
<td>6975 601 1.000 1.000</td>
<td>6975 335 1.000 0.558</td>
<td>7004 202 1.004 0.336</td>
<td>7053 133 1.011 0.221</td>
<td>8807 96 1.263 0.160</td>
</tr>
<tr>
<td>FPGA-12</td>
<td>3837 824 1.000 1.000</td>
<td>3837 407 1.000 0.494</td>
<td>3723 260 0.970 0.316</td>
<td>3774 168 1.028 0.205</td>
<td>3943 168 1.028 0.205</td>
</tr>
</tbody>
</table>

Geo. Mean - - 1.000 1.000 - - 0.999 0.355 - - 1.024 0.227 - - 1.065 0.169

† WLRT: Wirelength ratio compared to 1-thread execution.
† RT: Runtime ratio compared to 1-thread execution.

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

<table>
<thead>
<tr>
<th>Designs</th>
<th>8 Threads</th>
<th>16 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WL RT WLR</td>
<td>WL RT WLR</td>
</tr>
<tr>
<td>FPGA-1</td>
<td>230 11 1.013 0.256</td>
<td>232 9 1.020 0.200</td>
</tr>
<tr>
<td>FPGA-2</td>
<td>448 19 1.024 0.261</td>
<td>453 15 1.037 0.207</td>
</tr>
<tr>
<td>FPGA-3</td>
<td>1729 57 1.034 0.247</td>
<td>1474 46 1.007 0.219</td>
</tr>
<tr>
<td>FPGA-4</td>
<td>3406 68 1.009 0.270</td>
<td>3379 51 1.001 0.204</td>
</tr>
<tr>
<td>FPGA-5</td>
<td>6413 85 0.986 0.290</td>
<td>6549 64 1.007 0.217</td>
</tr>
<tr>
<td>FPGA-6</td>
<td>3125 114 0.994 0.254</td>
<td>3203 91 1.019 0.203</td>
</tr>
<tr>
<td>FPGA-7</td>
<td>6160 115 1.053 0.258</td>
<td>6507 90 1.112 0.201</td>
</tr>
<tr>
<td>FPGA-8</td>
<td>5848 130 1.053 0.247</td>
<td>6151 103 1.107 0.197</td>
</tr>
<tr>
<td>FPGA-9</td>
<td>7576 156 1.035 0.294</td>
<td>7341 114 1.003 0.197</td>
</tr>
<tr>
<td>FPGA-10</td>
<td>3043 150 0.994 0.258</td>
<td>3036 116 0.992 0.200</td>
</tr>
<tr>
<td>FPGA-11</td>
<td>7777 154 1.015 0.256</td>
<td>7287 113 1.045 0.189</td>
</tr>
<tr>
<td>FPGA-12</td>
<td>3845 187 1.002 0.227</td>
<td>3897 144 1.016 0.175</td>
</tr>
</tbody>
</table>

Geo. Mean - - 1.017 0.257 - - 1.030 0.199

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

<table>
<thead>
<tr>
<th>Components</th>
<th>1 Threads</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
<th>16 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Norm. RT</td>
<td>Norm. RT</td>
<td>Norm. RT</td>
<td>Norm. RT</td>
<td>Norm. RT</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Solve Linear System</td>
<td>0.851</td>
<td>0.449</td>
<td>0.231</td>
<td>0.125</td>
<td>0.076</td>
</tr>
<tr>
<td>Constr. Linear System</td>
<td>0.073</td>
<td>0.047</td>
<td>0.047</td>
<td>0.034</td>
<td>0.025</td>
</tr>
<tr>
<td>Rough Legalization</td>
<td>0.075</td>
<td>0.053</td>
<td>0.042</td>
<td>0.036</td>
<td>0.033</td>
</tr>
<tr>
<td>Partition Netlists</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PIPC</td>
<td>0.001</td>
<td>0.008</td>
<td>0.027</td>
<td>0.020</td>
<td>0.016</td>
</tr>
<tr>
<td>Others</td>
<td>0.001</td>
<td>0.1%</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

* 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.