ABCDPlace: Accelerated Batch-based Concurrent Detailed Placement on Multi-threaded CPUs and GPUs

Yibo Lin *Member, IEEE*, Wuxi Li *Member, IEEE*, Jiaqi Gu *Student Member, IEEE*,
Haoxing Ren *Senior Member, IEEE*, Brucek Khailany *Senior Member, IEEE*, David Z. Pan *Fellow, IEEE*

Abstract—Placement is an important step in modern very-large-scale integrated (VLSI) designs. Detailed placement is a placement refining procedure intensively called throughout the design flow, thus its efficiency has a vital impact on design closure. However, since most detailed placement techniques are inherently greedy and sequential, they are generally difficult to parallelize. In this work, we present a concurrent detailed placement framework, *ABCDPlace*, exploiting multithreading and GPU acceleration. We propose batch-based concurrent algorithms for widely-adopted sequential detailed placement techniques, such as independent set matching, global swap, and local reordering. Experimental results demonstrate that *ABCDPlace* can achieve $2 \times -5 \times$ faster runtime than sequential implementations with multi-threaded CPU and over $10 \times$ with GPU on ISPD 2005 contest benchmarks without quality degradation. On larger industrial benchmarks, we show more than $16 \times$ speedup with GPU over the state-of-the-art sequential detailed placer. ABCDPlace finishes the detailed placement of a 10-million-cell industrial design in one minute.

I. INTRODUCTION

Placement is a critical stage in the VLSI design flow. It determines the physical locations of logic gates (cells) in the circuit layout and its solution has a huge impact on the subsequent routing and post-routing closure. Placement usually consists of three stages: global placement, legalization, and detailed placement. Global placement provides rough locations of standard cells. Legalization then removes overlaps and design rule violations based on the global placement solution. In the end, detailed placement incrementally improves the solution quality. In the VLSI design iterations, detailed placement may be invoked many times to recover the solution quality from post-placement perturbations, such as buffer insertion, routability and timing optimization. Therefore, the efficiency and quality of detailed placement play an important role in speeding up the design iterations.

Detailed placement widely involves combinatorial optimization, graph algorithms, and greedy heuristics. Various effective algorithms have been proposed with the strategy of extracting a subset of cells and exploring the corresponding solution space iteratively [1]–[11], including independent set matching, global swap, local reordering, and row-based refinement, etc. These algorithms are usually designed for sequential execution on single-threaded machines. They are very difficult to be parallelized.

With the increasing design scale and complexity, sequential algorithms are encountering challenges in efficiency due to tight time-to-market budgets. They are becoming bottlenecks that hinder the turn-around time. As most of the recent research efforts for detailed placement have been switched to incorporating new objectives and constraints, such as routability, mixed-cell-height designs, manufacturing constraints [12]–[17], there is little progress on the core detailed

Y. Lin is with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing, China.

W. Li is with Xilinx Inc., CA, USA.

J. Gu and D. Z. Pan are with The Department of Electrical and Computer Engineering, The University of Texas at Austin, TX, USA.

H. Ren and B. Khailany are with NVIDIA Corporation, Austin, TX, USA.

placement problem for wirelength minimization. Figure 1 roughly sketches the runtime scaling trends for recent placers. While the comparison may not be fair due to different objectives and constraints during optimization, it still shows the runtime of placement engines has not been improved even with more and more powerful CPUs. NTUplace3 [6] proposed in 2008 is still competitive in efficiency for wirelength optimization, especially that it is widely used in the most recent placement researches [18]–[21].

With modern computing platforms like multi-core processors and graphic processing units (GPUs), massively parallel computing has the potential to accelerate the placement optimization. So far, a majority of the literature has been exploring global placement acceleration or simulated annealing-based approaches [22]–[27]. The recent study from DREAMPlace [28] demonstrated that after accelerating global placement, detailed placement becomes the runtime bottleneck by taking more than 75% of the entire placement time for a benchmark with 2M cells. Thus, accelerated detailed placement engines are urgently desired to further speedup the flow. However, the greedy and iterative nature of existing detailed placement techniques raise the bar of effective parallelization. There is limited prior work investigating the potential of massive parallelization for detailed placement techniques. Only recently, Dhar et al [29] have explored multithreading and GPU acceleration for a row-based interleaving algorithm in FPGA placement. This is far from enough, as effective detailed placement engines usually consist of multiple techniques and typically involve graph algorithms and greedy heuristics.

With increasingly powerful multi-core processors and GPUs, parallel computing has demonstrated its efficiency in solving large graph problems [30]–[32]. As detailed placement heavily involves graph traversal and analytics, there is a high potential to accelerate detailed placement algorithms with massive parallelization as well. Therefore, in this work, we present *ABCDPlace*, an open-source GPU-accelerated detailed placement engine leveraging batch-based concurrency. We re-design the widely-adopted detailed placement techniques and propose parallel versions for multi-threaded CPUs and GPUs. The main contributions of the paper are summarized as follows.

- We propose an open-source batch-based concurrent detailed placement framework, *ABCDPlace*, with multithreading and GPU acceleration.
- We propose parallel detailed placement algorithms for widely-adopted sequential techniques, such as independent set matching, global swap, and local reordering, leveraging batch execution.
- Experimental results demonstrate that, compared with a highly efficient sequential detailed placer NTUplace3 [6], ABCDPlace is able to achieve over $10 \times$ and $16 \times$ speedup with GPUs on ISPD 2005 contest benchmarks and industrial benchmarks, respectively, without quality degradation. The multi-threaded CPU version also achieves around $2 \times -5 \times$ speedup. Experiments on ISPD 2015 contest benchmarks also indicate that our placer does not degrade the global routing congestion.

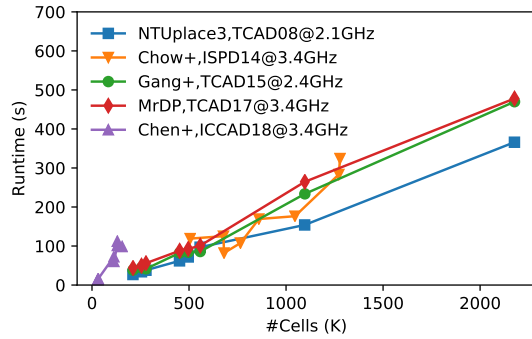*ABCDPlace* has been integrated into *DREAMPlace 2.0* as the default

Fig. 1: Rough runtime scaling for the recent development of detailed placement engines [6], [12], [13], [17], [33]. The runtime values are collected from the papers with the CPU frequencies shown in the legend, except for NTUplace3 [6] where we ran the experiments with the binary release.
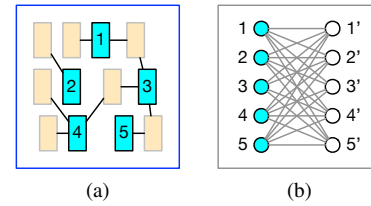

Fig. 2: Independent set matching. (a) A set of independent cells within a window. (b) Construction of a bipartite graph to find the best assignment.
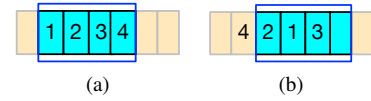

Fig. 3: Example of local reordering with sliding window. (a) Sliding window at one step; (b) next step.

detailed placement engine released on Github [1]. Each algorithm is implemented as an operator that can be invoked with the Python API in DREAMPlace. The rest of the paper is organized as follows. Section II introduces the background of detailed placement algorithms and the problem formulation. Section III explains the parallel algorithms in details. Section VI validates the algorithms with experimental results. Section VII concludes the paper.

II. PRELIMINARIES

Detailed placement typically assumes a legal initial placement is given and performs incremental refinement. The main objective is usually half-perimeter wirelength (HPWL), which is computed as the bounding box of each net as follows,

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

where e represents a net (hyperedge) in a set of nets E and i, j represent any of two cells connected in e. The output of detailed placement is a legal placement solution with optimized wirelength. In general, a detailed placer often runs several key strategies to explore different solution spaces iteratively. For example, FastPlace serializes iterations of global swap and local reordering until no significant wirelength improvements occur [2], [11], [34]. NTUplace serializes iterations of local reordering (branch-and-bound cell swap [6]) and independent set matching [6], [35]. Each of these strategies extract a small set of cells and perturb them to find a better solution. In this work, we focus on the parallelization of the following three strategies: (1) independent set matching; (2) global swap; (3) local reordering.

A. Independent Set Matching

The previous two strategies only consider perturbations among very few cells, e.g., 2-4. Independent set matching explores another solution space that involves more cells [6]. Figure 2 shows an example, where cells not connected to each other within a window are extracted. As these cells form an independent set, movement of one cell will not affect the wirelength of other cells in the set. We can independently pre-compute the incident wirelength change caused by assigning one cell to the location of another cell in the set and find the optimal assignment by solving a maximum weighted

[1] https://github.com/limbo018/DREAMPlace

bipartite matching problem. Sometimes one cell cannot be assigned to the location of another cell due to lack of space. In this case, no edge will be added between that cell and location in the bipartite graph. The independent set is created by searching for unconnected cells greedily in a window. Due to the connectivity between cells, sequential implementation is again a natural choice.

B. Global Swap

A general description of global swap is to repeat the following process: pick a cell i; find another cell or space j in a search region that maximally improves the wirelength after swap; then swap the two cells. There are various heuristics to determine the search region for a cell. It can be the bin in which the cell is located or the optimal region of the cell [2]. Considering that a subsequent cell movement is dependent to the previous cell movements due to the connectivity and potential overlap issue, this process is usually performed sequentially.

C. Local Reordering

Local reordering shuffles a sequence of consecutive cells for the best permutation [2], [6], as shown in Figure 3. It is a window-based strategy that works on k cells at each step and repeats by sliding the window from left to right. As the number of permutation for a sequence of k is $k!$, it is only affordable to use a small value of k, e.g., 3 or 4. When k goes down to 2, this local reordering step becomes a special case of global swap. As the sliding windows have to overlap for large enough solution space and cells are connected, most of the existing implementations are sequential.

D. Problem Formulation

In this work, we aim at accelerating the detailed placement techniques: global swap, local reordering, and independent set matching, with massive parallelization on multi-threaded CPUs and GPUs. Our framework, *ABCDPlace*, takes a netlist with a legal placement solution as input, performs incremental wirelength optimization, and outputs an optimized legal solution. The placer runs on both CPUs and GPUs, and leverages parallel computing techniques to speed up the sequential algorithms. Our main technique to improve the parallelizability is to explore batch execution from the original sequential procedures. The details will be covered in Section III.
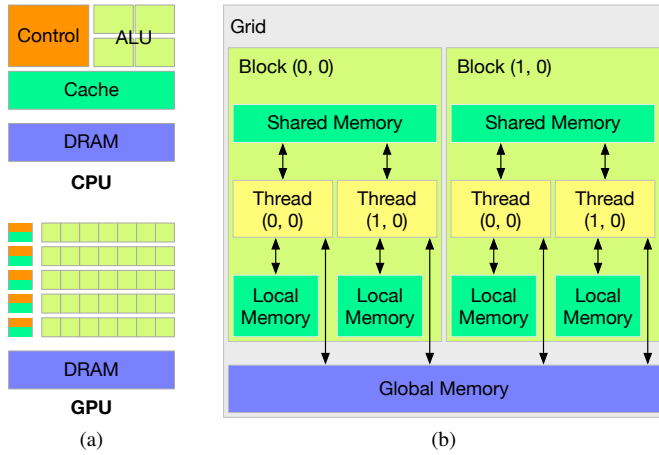
Fig. 4: (a) Comparison between CPU and GPU architectures. (b) Computation units and memory hierarchy on GPU.

E. GPU Architecture and Programming

GPU programming is quite different from CPU due to the discrepancy in architectures and programming models. Figure 4 compares the architectures for CPU and GPU. A CPU is roughly composed of a control unit, computation units (ALU), cache, and memory (DRAM). A GPU also has such components, but with very different scale and performance. It consists of a grid of computation units with simple control units and small cache, which indicates that a GPU prefers parallel execution of small tasks with simple control flows. Each computation unit on a GPU may not be as powerful as that on CPU, but due to massive parallelization, it can potentially be faster.

Unlike relatively mature programming models on CPU, GPU programming still requires careful design of both algorithms and implementations. The performance is likely to be much slower than CPU even for a fully parallelizable task due to poor implementations. The reason mainly comes from the flexible configurations to the computation units at runtime. The computation units on a GPU can be viewed as a grid of blocks. Each block consists of many threads (at most 1024 for most GPUs). A block can be assigned with a piece of shared memory that can be accessed by its threads more efficiently than the global memory. However, there is an upper limit to the total amount of the shared memory for all blocks, e.g,. 48∼96KB according to GPU devices. Moreover, thread synchronization within a block is much cheaper than the device-level synchronization. To perform computation on a GPU, a program on the host CPU needs to launch a kernel function call with the configurations of blocks, threads, and shared memory. Such a function call has an overhead around several micro seconds. In other words, frequent kernel calls are not preferred in GPU programming.

With all these differences in the hardware architectures and programming models, straightforward parallelization schemes on CPUs often do not work on GPUs. In other words, GPUs require specially designed algorithms and threading schemes to demonstrate the power of massive parallelization.

III. The ABCDPlace Algorithms

This section explains the details on concurrent versions of independent set matching, global swap, and local reordering.

Algorithm 1 Sequential Independent Set Matching

Require: A circuit netlist $G = (V, E)$, locations of cells, and maximum size of an independent set L;
Ensure: Minimize wirelength by independent set matching;
1: **for** each cell $v \in V$ **do**
2:    Search independent cells with the same sizes as v in the neighborhood and form an independent set I, $s.t., |I| \le L$;
3:    Compute costs of permuting cell locations in I;
4:    Solve the LAP with the weights;
5:    Apply the assignment solution;

A. Concurrent Independent Set Matching

Algorithm 1 sketches a rough procedure for independent set matching according to [6]. The word *independent* describes cells not connected to each other. Thus, the movement cost for a cell in an independent set can be computed without considering the locations of other cells in the set. With an independent set, we can construct a bipartite graph and solve the linear assignment problem (LAP) for the best locations of cells in the set. The algorithm follows four steps iteratively: 1) extract an independent set; 2) compute permutation costs, which means the cost of moving one cell to the location of another one in the independent set I; 3) solve the LAP; 4) move the cells according to the solution of LAP. The algorithm is very difficult to parallelize following the same procedure, because the maximum size of independent set L is usually limited to around 100. Even though the cost computation step for an $L \times L$ matrix can be parallelized, the bulk runtime actually comes from independent set extraction and LAP solving, which are typically sequential.

To improve the parallelization, we design a concurrent independent set matching algorithm. Although the algorithm still runs in iterations, it converges much faster than iterating through all cells like Algorithm 1. The major advantage lies in the fully parallelizable internal steps. Figure 5 provides an intuitive explanation of the steps. We first extract a maximal independent set [2] with given seeds, which are likely to contain tens of thousands of cells. The set is then partitioned into many small subsets such that physically close cells with the same sizes are in the same subsets. Next, we can solve the LAP instances for all the subsets in the batch independently after computing the costs of cell permutation. In the end, the solutions are applied in parallel as well. The rest of the section will cover the non-trivial parallel implementation of these steps, including parallel maximal independent set extraction, parallel partitioning, and batch LAP solving.

1) Parallel Maximal Independent Set Extraction: A sequential maximal independent set algorithm follows the procedure:

- For each node in the graph:
-     If not in the set yet:
-         Add to the set and remove all its neighbors in the graph.

This algorithm has a time complexity of $\mathcal{O}(|V|)$ if $|E|$ is of the same magnitude as $|V|$, as it needs to traverse the entire graph once. The algorithm becomes slow with large graph sizes. There exist parallel maximal independent set algorithms that can achieve $\mathcal{O}(\log^2 |V|)$ time complexity, e.g., Blelloch's Algorithm [37].

Algorithm 2 describes a parallel maximal independent set algorithm based on the Blelloch's algorithm [37] that is suitable for our application. The algorithm is general enough to handle hypergraphs as well. Lines 4 to line 7 ensure that only the vertex v with lowest order $R(v)$ among its neighbors joins I. As the vertex with the globally smallest R value will always join I, the algorithm guarantees to

---

[2] A maximal independent set is different from a maximum independent set. The former can be found greedily, while the latter is NP-complete [36].
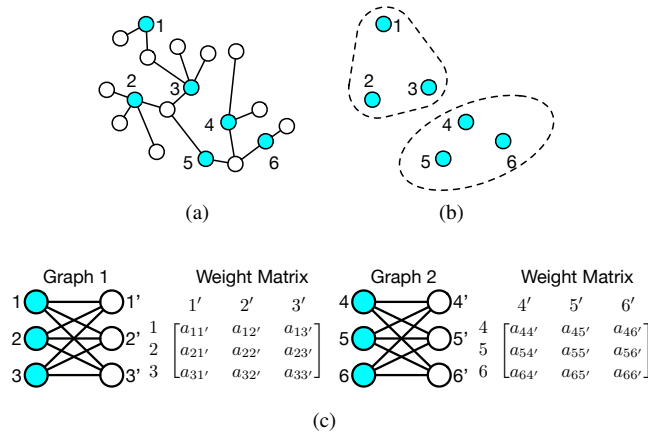
Fig. 5: Steps for batch-based concurrent independent set matching. (a) Maximal independent set extraction. (b) Independent set partitioning. (c) Linear assignment solving for a batch of bipartite graphs.

Algorithm 2 Parallel Maximal Independent Set Algorithm

Require: A graph $G = (V, E)$, a random order $R, s.t., |R| = |V|$;
Ensure: A maximal independent set I contains $v_{\arg\min R} \in V$;
1: $I \leftarrow \emptyset$;
2: **while** V is not empty **do**
3:     **for** each $v \in V$ **do**   ▷ Parallel kernel
4:         **if** $R(v) < R(w), \forall (v, w) \in E$ **then**
5:             $I \leftarrow I \bigcup \{v\}$;   ▷ Add to I
6:             $G \leftarrow G \setminus v$;   ▷ Remove v from G
7:             $G \leftarrow G \setminus w, \forall (v, w) \in E$;   ▷ Remove v's neighbors

---

make progress in each round (line 3 to line 7). It will take at most $\mathcal{O}(\log^2 |V|)$ rounds. Meanwhile, the task within each round is fully parallelizable, as each vertex can be processed independently within the for loop. In practice, early exit is possible if enough vertices are collected for solving LAP. The random order sequence R can also be generated efficiently with a parallel random shuffling of the sequence $0, 1, \ldots, |V|$.

2) Parallel Partitioning with Balanced K-Means Clustering: The maximal independent set is too large for LAP algorithms. Observing that most of the cell movement happens locally, it is good to partition the independent set into a batch of small subsets such that cells in the same subset are physically close to each other. A sequential implementation might be distributing the cells into bins and performing spiral walk to greedily add nearby cells to subsets. This is not runtime-efficient for GPU because of its sequential nature and irregular memory access patterns in spiral search, while on CPU, the runtime is acceptable and not the bottleneck. Therefore, we adopt K-Means clustering for GPU in this step, leveraging parallel reduction to find the closest centroids for clusters. The conventional objective for K-Means clustering is to find K centers,

$$\min \quad \sum_{i=1}^{K} \sum_{x \in S_i} \|x - \mu_i\|^2, \tag{2}$$

where μ denotes the centers. However, such an objective may result in imbalanced clusters. This is not preferred for parallel solving of LAP instances, especially for GPU acceleration. To achieve balanced clustering results, we consider a weighted objective,

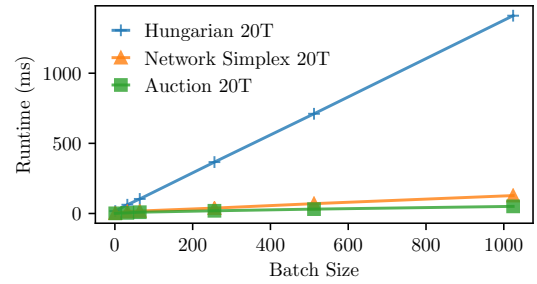$$\min \quad \sum_{i=1}^{K} \sum_{x \in S_i} w_i \|(x - \mu_i)\|^2, \tag{3}$$

Fig. 6: Runtime comparison of batch solving for LAPs on CPU. "20T" stands for 20 threads on CPU.

where the weight w_i for each cluster is adjusted in each iteration to penalize large clusters. Given a target cluster size s_t, weights are initialized to 1 and empirically updated at the kth iteration as follows,

$$w_i^{k+1} \leftarrow w_i^k \times (1 + 0.5 \cdot \log(\max\{1, |S_i|/s_t\})). \tag{4}$$

Intuitively, the weight of a cluster increases if its size goes beyond s_t. This is an empirically determined function. Other functions with similar trends may also work. In the experiment, the number of clusters K is computed as the ratio of the number of nodes in a maximal independent set over the expected cluster size s_t, where $s_t = 128$. We observe that 2 iterations of K-Means have already achieved reasonable partitioning results with rather balanced distribution of subsets.

3) Batch Solving for Linear Assignment Problems: LAP can be solved with many algorithms, such as Hungarian algorithm, network flow algorithms, auction algorithm, etc. The mathematical formulation can be written as,

$$
\begin{aligned}
\max \quad & \sum_{i,j} a_{ij} x_{ij}, \\
\text{s.t.} \quad & \sum_{i=1}^{N} x_{ij} = 1, \quad j = 1, 2, \ldots, N, \\
& \sum_{j=1}^{N} x_{ij} = 1, \quad i = 1, 2, \ldots, N, \\
& x_{ij} \geq 0, \quad i, j = 1, 2, \ldots, N,
\end{aligned}
\tag{5}
$$

where $x_{ij} = 1$ indicates assigning i to j, and a_{ij} is the weight of such an assignment. The Hungarian and network flow algorithms are widely adopted as sequential solvers [6], [33], while auction algorithms [38] are generally the choice for distributed computing platforms due to its easy-to-parallelize nature [32], [39].

Figure 6 shows a comparison on solving a batch of LAP instances with different algorithms on a multi-threaded CPU. Hungarian adopts the implementation from [40] and network simplex (a highly efficient network flow algorithm in practice [33]) uses the solver from Lemon [41]. The auction algorithm adopts our own implementation described in Algorithm 3. Each LAP instance is solved with a single thread. We can see that network simplex is much faster than the Hungarian algorithm, and our auction algorithm can achieve further 2× speedup over network simplex. While the solvers are mostly treated as black boxes in placement algorithms, it is sometimes necessary to study the details for acceleration targeting specific hardware platforms.

Auction algorithms consider the problem in which N persons (cells in this work) bid for N items (locations in this work) with a weight of a_{ij} (negative wirelength cost of assigning a cell to a location as the algorithm maximizes the objective). A typical auction algorithm repeats a bidding phase and an assignment phase that are fully parallelizable,

as shown in Algorithm 3. In the bidding phase from line 5 to 9, each person finds the item j^* with the largest $a_{ij} - price_j$ value recorded in the temporary variable v_{ij*} and the second largest $a_{ij} - price_j$ value recorded in the temporary variable w_{ij*}, respectively. Then the price increment for bidding is computed as bid_{ij*} and item j^* is marked in $sbid_{j*}$. In the assignment phase from line 10 to 16, each bid item looks for bidder i^* with the highest bidding price increment b_{i*}, assigns itself to person i^*, and raises its price by b_{i*}. Looping between these two phases will lead to an optimal assignment solution with the maximum objective. The auction epsilon ϵ, which indicates the price augmentation step, controls the numerical precision of convergence to the optimal solution. Parallelization of Algorithm 3 can be realized by parallelizing the bidding and assignment phases. For example, one thread can be allocated to work on each person independently in the bidding phase (line 5 to line 9); we can also have one thread work on each item independently in the assignment phase as well.

Efforts have been spent on accelerating LAP algorithms with large N (> 1000). However, in this problem, cells only need local movements, so each LAP instance is small with N around 100. There are many such small instances [6]. Our simple experiment on an LAP with $N = 128$ shows that the GPU implementation requires around 5ms, while a single-threaded CPU implementation takes around 17ms, 2ms, 1ms using Hungarian algorithm [40], network simplex [41], and auction algorithm (equivalent to the results of batch size 1 in Figure 6), respectively, making GPU unable to compete with CPUs.

Therefore, a batch-based auction algorithm is required to solve multiple LAP instances with the same problem size simultaneously with massive parallelization on GPUs. A naive way for batch execution is to adopt CUDA's multi-stream scheme by assigning each LAP to one CUDA stream. Unfortunately, multi-stream is usually inefficient for thousand of streams due to crowded kernel launches and we even observe longer runtime with that. Hence, we propose a GPU implementation specifically optimized for batch solving of small LAP instances, as shown in Figure 7 and Algorithm 4. To mitigate the expensive communication and synchronization overhead between CPUs and GPUs, the batch-based auction algorithm integrates the entire while loop into a CUDA kernel and assigns each LAP instance to one thread block. By doing so, expensive device-wide synchronization is minimized and the number of kernel launches is reduced from $\mathcal{O}(BK)$, empirically around 1 million, down to one, where B is the batch size and K is the largest number of iterations for one LAP instance in the batch, usually larger than 1000. This implementation is specifically designed for small LAP instances because the maximum number of threads in one block is typically limited to 1024, which is a constraint from GPUs. Although device-wise synchronization is no longer required, block-wise synchronization is still needed, as shown in line 8 and line 10 of Algorithm 4, which can be invoked within a kernel. Threads within a thread block need to wait for all their tasks finished before moving forward. Furthermore, we introduce an annealing scheme for ϵ to speed up the convergence by solving the LAP instance from a large ϵ_{max} to small ϵ_{min} and use the prices of previous solving as the starting point for the next, as described in line 5 and line 13 of Algorithm 4. In the experiment, we set $\epsilon_{max} = 10, \epsilon_{min} = 1, \gamma = 0.1$. We can improve the runtime on GPU by more than $100\times$ over the multi-stream implementation for typical batch size of 1024 and $N = 128$.

B. Concurrent Global Swap

Global swap is another widely-adopted placement technique [2], [4]. Without loss of generality, a typical procedure of global swap is shown in Algorithm 5. It consists of five major steps and iteratively

---

Algorithm 3 Auction Algorithm for One LAP Instance

Require: An $N \times N$ weight matrix A for LAP and auction ϵ;
Ensure: Find the assignment solution with maximum objective;
1: Define $price$ as a length-N array, initialized to 0;
2: Define bid as an $N \times N$ matrix, $sbid$ as a length-N array;
3: **while** not all items assigned **do**
4:     $bid_{ij} \leftarrow 0, \forall i, j; sbid_j \leftarrow 0, \forall j;$
5:     **for** each person i **do**   ▷ Parallel bidding kernel
6:       $v_{ij*} \leftarrow \max_j(a_{ij} - price_j);$ ▷ Largest
7:       $w_{ij*} \leftarrow \max_{j \neq j*}(a_{ij} - price_j);$ ▷ Second largest
8:       $bid_{ij*} \leftarrow v_{ij*} - w_{ij*} + \epsilon;$
9:       $sbid_{j*} \xleftarrow{atomic} 1;$
10:    **for** each item j **do**   ▷ Parallel assignment kernel
11:      **if** $sbid_j$ **then**
12:        $b_{i*} \leftarrow \max_i bid_{ij};$
13:        **if** item j has been assigned **then**
14:          Unassign j;
15:        $price_j \leftarrow price_j + b_{i*};$
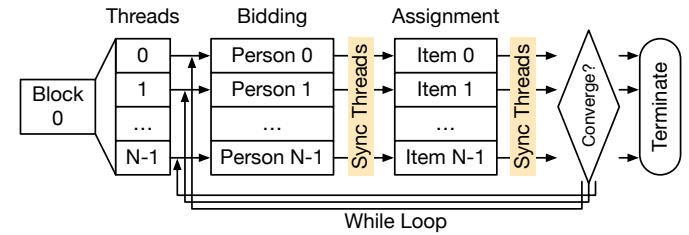16:   Assign item j to person i^*;

Fig. 7: Parallelization scheme for batch-based auction algorithm. One thread block is used to solve one LAP with the while loop within the kernel.

runs for each cell. There are various heuristics to compute the search region, such as directly using the bin in which a cell is located or its optimal region [2]. Moreover, during the sequential search for best swap candidates, we can start from potentially good regions to bad regions, such that early exit is possible if a candidate has been found.

Figure 8a shows the runtime breakdown of a sequential implementation of Algorithm 5 running on CPUs. The runtime portion for ApplyCand is not shown as it is too small. The plot indicates that CalcSwapCosts takes the majority of the runtime. Naive parallelization of CalcSwapCosts does not lead to much speedup as for each cell, we do not look for a large enough number of candidate

---

Algorithm 4 Batch-based Auction Algorithm for LAP instances

Require: A $B \times N \times N$ weight tensor, $\epsilon_{max}, \epsilon_{min}, \gamma$;
Ensure: Find $B \times N$ assignment matrix with maximum objectives;
1: Define $price$ as a $B \times N$ matrix, initialized to 0;
2: Define bid as a $B \times N \times N$ tensor, $sbid$ as a $B \times N$ matrix;
3: $\epsilon \leftarrow \epsilon_{max};$ ▷ Parallel kernel begins
4: $bid \leftarrow blockIdx, tid \leftarrow threadIdx;$
5: **while** $\epsilon \geq \epsilon_{min}$ **do**
6:    **while** not all items assigned for LAP instance bid **do**
7:      Initialize bid and $sbid$;
8:      Synchronize threads;
9:      Bidding phase for person tid with ϵ;
10:     Synchronize threads;
11:     Assignment phase for item tid;
12:     Synchronize threads;
13:   $\epsilon \leftarrow \gamma\epsilon;$ ▷ Parallel kernel ends
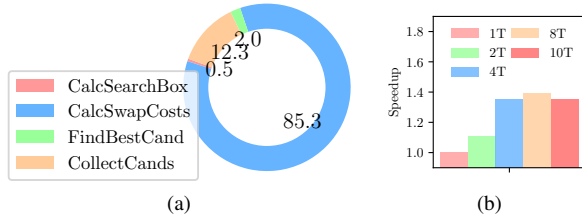

Fig. 8: (a) Runtime breakdown of sequential global swap implementation on CPU for `bigblue4` with 2M cells. (b) `CalcSwapCosts` speedup with naive multithreading. "T" stands for threads on CPU.

Algorithm 5 Sequential Global Swap Algorithm

Require: A circuit netlist $G = (V, E)$ and locations of cells;
Ensure: Minimize wirelength by swapping cells;
1: **for** each cell $v \in V$ **do**
2:     Compute search region R for v;   ▷ CalcSearchBox
3:     Collect swap candidates C in region R;   ▷ CollectCands
4:     Compute swap costs for candidates;   ▷ CalcSwapCosts
5:     Find candidate c^* with minimum cost;   ▷ FindBestCand
6:     Apply the swap with the candidate c^*;   ▷ ApplyCand

---

cells for swapping and the threading overhead is not affordable. As shown in Figure 8b, our experiments on `bigblue4` actually show the speedup to `CalcSwapCosts` quickly saturates to $1.4\times$ even with 10 threads enabled, when we use `OpenMP` to parallelize the *for* loop for swap cost computation. Especially when considering that GPUs have low single-threaded performance, but with thousands of threads available, creating enough parallel tasks for `CalcSwapCosts` to hide latency is essential to good speedups with GPU acceleration.

To apply task decomposition, we develop a batch-based concurrent global swap to improve the performance of parallelization. Figure 9 explains the intuition. Instead of processing one cell each time as in Algorithm 5, processing a batch of cells explores more parallelism for `CalcSwapCosts` and thus is potentially beneficial. The overall algorithm of the concurrent global swap is presented in Algorithm 6. We precompute the search regions for all cells in parallel in line 1 and line 2. From lines 3-7, we fetch one batch of cells every time and perform concurrent global swapping. Suppose that there are B cells in a batch and on average we check 100 swap candidates for each cell, there will be $100 \times B$ concurrent tasks, which is enough for relatively high occupancy of GPU resources. This batch-based concurrency applies to both `CollectCands` and `CalcSwapCost`. We will discuss the aforementioned functions one-by-one in the rest of the section.

Figure 10 shows the parallel implementation of `CollectCands` conceptually. We allocate a fixed number of thread blocks for each cell in the batch. Each thread will collect one candidate cell for the candidate array. The candidate array is pre-allocated with a fixed

---

Algorithm 6 Concurrent Global Swap Algorithm

Require: A circuit netlist $G = (V, E)$, locations of cells, batch size B;
Ensure: Minimize wirelength by swapping cells;
1: **for** each cell $v \in V$ **do**   ▷ Parallel
2:     $R(v) \leftarrow$ CalcSearchBox(v);
3: **for** each batch of cells $B_v \in V$ **do**
4:     $B_C \leftarrow$ CollectCands(B_v, R);   ▷ Parallel
5:     CalcSwapCosts(B_C);   ▷ Parallel
6:     $B_{c^*} \leftarrow$ FindBestCand(B_C);   ▷ Parallel Reduction
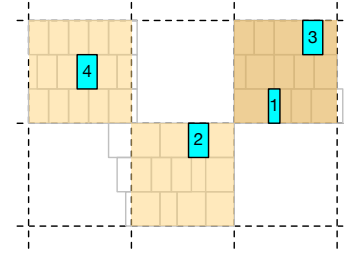7:   ApplyCand(B_{c^*});   ▷ Sequential

---

Fig. 9: Batch-based concurrent global swap. Cells in the batch and regions to search for candidates are highlighted.
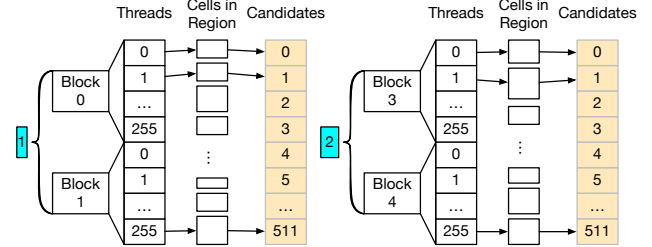

Fig. 10: Parallelization scheme for `CollectCands`.

maximum number of candidates. The runtime of this step can be significantly improved since the memory access pattern is rather regular. Besides, the cost computation for all candidates can be done in parallel as well. Due to the large number of candidates, i.e., batch size $\times$ maximum number of candidates per cell, the workload can be distributed to many threads for speedup. The details on batch sizes and maximum number of candidates per cell are discussed in the following paragraph.

After parallelizing the candidate collection and cost computation, the swap costs are stored in a $B \times K$ matrix-like structure with B as batch size and K as maximum number of candidates for each cell. Best candidates with minimum costs can be selected by using parallel reduction operations [42]. Given P threads, the time complexity is $\mathcal{O}(\frac{BK}{P} + \log K)$ [43], where B is around 32 to 256, K is around 512 to 1024 in the experiments, and P is in thousands for GPUs.

The last step is to apply swapping to the best candidates, shown as function `ApplyCand`. As the cells and candidates in the previous steps may have dependency to each other, this is the step we resolve such dependency issues. For a given batch, there are at most B candidates that need to be applied. There might be conflicts between candidates. For example, two cells in the batch may tend to swap with the same cell, which will result in incorrect costs if both swaps applied. To resolve such a data race, sequential execution is adopted. We give up candidates whose cells to swap with have been moved or other cells connected to these two cells have been moved in this batch. Note that this step is not the runtime bottleneck even with sequential execution.

In the experiment, the search region for one cell is set to one bin, whose width and height are around 3-row height. We can control the batch size for the efficiency and resource usage. With larger batch sizes, the efficiency may improve, but require more GPU memories, as all the storage needs to be pre-allocated. We observe the speedup starts to saturate when the batch size is around 256, so we adopt this value.
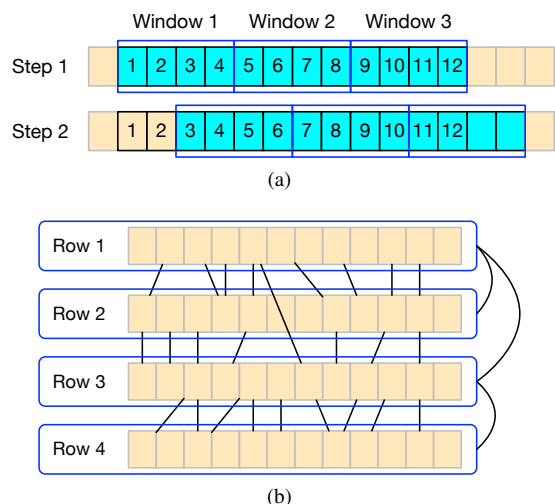
Fig. 11: Explore parallelization in local reordering. (a) Parallel sliding windows within a row. (b) Groups of independent rows ({row 1, row4}, {row 2}, {row 3}).

C. Concurrent Local Reordering

Different from global swap in which a cell may search for candidates quite far away from its own location, local reordering works on a very small sliding window with k cells in a row. The parameter k is small due to the $k!$ number of possible permutations for enumeration.

1) Parallel Enumeration: One straightforward multithreading scheme is parallel enumeration of the $k!$ permutations. There are two issues for this scheme: i) for multi-threaded CPU, the task of enumerating one permutation is too small to compensate the threading overhead; ii) for GPU acceleration, the number of permutations like $3! = 6$ and $4! = 24$ are not enough to fill thousands of GPU threads. Therefore, parallel enumeration itself is not enough to boost the efficiency. We need to find more parallel tasks by exploring multiple dimensions of parallelism.

2) Parallel Sliding Windows: The original sequential algorithm slides a window from left to right and solves one window at a time. We consider the potential of solving multiple windows at the same time, as shown in Figure 11a, where window 1, 2, 3 can be solved in parallel. However, one may argue the connectivity of cells between windows is likely to cause suboptimality, e.g., cell 6 connects to cell 2 and their locations are undecided during the solving. Actually, this will not affect the optimality for each permutation problem within a window, because the relative ordering of cells between different windows is fixed. More specifically, cell 6 knows cell 2 is always at its left side, and so as cell 2. To this end, when computing the HPWL of the net incident to both cell 2 and 6, cell 6 can assume cell 2 is located at the left boundary of cell 6's window. Similarly, cell 2 can assume cell 6 is located at the right boundary of cell 2's window. This trick is widely-used in ordered row placement [2] and the optimality within each window still holds.

Solving parallel sliding windows once cannot cover enough solution space as explored by sequential sliding. The parallel solving needs to be conducted multiple times, as shown in Figure 11a. After step 1, step 2 shifts all the parallel windows with an offset and performs another round of solving. We can control the step size for the offset for a reasonable number of rounds. In the experiment, we set the step size as $\frac{k}{2}$.
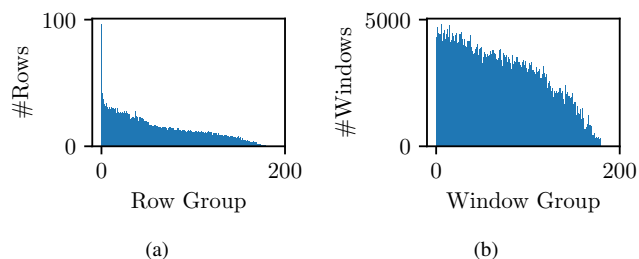

Fig. 12: (a) Group sizes of independent rows; (b) group sizes of independent windows on `bigblue4`.

3) Independent Rows: We further investigate possible parallel tasks by extracting independent rows, as shown in Figure 11b. Independent rows refer to a group of rows that do not have any two cells share a common net. In the figure, we can find three such groups of {row 1, row4}, {row 2}, {row 3}. This approach is valid under the assumption that most connections are local in the detailed placement problem.

With the aforementioned three dimensions of parallelism, a significant number of parallel tasks can be performed. Figure 12 shows the distribution of group sizes at row group level and window group level on `bigblue4`. Row group refers to the grouping of independent rows. Window group refers to the incorporation of independent rows and parallel sliding windows. All the instances within a group can be solved in parallel. Hence, we can see the effectiveness in finding independent tasks.

IV. SUMMARY OF PARALLEL CPU AND GPU IMPLEMENTATIONS

To achieve high efficiency on both multi-threaded CPU and GPU, we optimize the implementations separately with slightly different threading strategies for the three algorithms. We summarize the major differences in Table I and in this section.

1) Concurrent Independent Set Matching: Both multi-threaded CPU and GPU versions implement the same parallel maximal independent set algorithm, but the single-threaded CPU adopts the greedy sequential algorithm mentioned at the beginning of Section III-A1, because the sequential algorithm can finish the extraction in one iteration, while the parallel algorithm requires multiple iterations.

Another main difference lies in the partitioning step, where the CPU version adopts sequential spiral search and the GPU version adopts the K-Means clustering, as spiral search is too expensive on GPUs and K-Means clustering is too expensive on CPUs. In the LAP solving step, each CPU thread solves one LAP instance, while the GPU version adopts the batch implementation discussed in Section III-A3. When applying the solutions, we use each CPU thread for one independent set, while each GPU thread only works on one cell in an independent set.

2) Concurrent Global Swap: For the CPU version, we allocate each thread for one cell during the batch execution of candidate collection, cost computation, and best candidate finding. As the typical batch size is 256 or larger, there are enough tasks for each CPU thread. For the GPU version, we allocate threads in a finer granularity. In candidate collection, each thread collects one candidate for a cell, as mentioned in Figure 10; in cost computation, each thread computes costs for one candidate; in finding the best candidates, parallel reduction is used.

3) Concurrent Local Reordering: For the CPU version, the parallelization is implemented at the level of independent rows. That is, within a group of independent rows, each thread will solve the enumeration problems sequentially by sliding windows in one row.

7

For the GPU version, we allocate each thread for each independent window with one permutation. Then parallel reduction is performed to find the best permutation for each window.

V. POSSIBLE EXTENSIONS

ABCDPlace aims at accelerating the fundamental wirelength optimization techniques in detailed placement. In modern design flow, detailed placement sometimes also needs to consider other objectives such as routability and timing. The parallelization strategies developed in this work can be extended to handle these objectives.

For routability optimization, a typical way for extension is to add overflow penalty to the objective along with the wirelength cost. One example is the NTUplace4 series [5], [44]. As the routing or density overflow map can be precomputed, the overflow penalty for the movement of an individual cell can be calculated incrementally. Then, a weighted sum of overflow penalty and wirelength cost can guide the detailed placement engine to optimize routing congestion. For timing optimization, typical techniques include net weighting and incorporation of extra timing cost into the objective [45], [46]. An external timing analysis engine can help achieve this goal.

Therefore, extensions of these detailed placement techniques for routability and timing optimization in general involve in integrating other penalty terms into the wirelength cost, while the skeletons of the algorithms remain similar. This work can provide practical insights in developing algorithms for routability and timing optimization. We leave the incorporation of these objectives in the future.

Meanwhile, multiple-row height cells become common in modern designs. Our current implementations only work on single-row height cells and fix the multiple-row height ones. For independent set matching and global swap, we extend to handle multiple-row height cells if we work on cells with the same sizes. We plan to incorporate this feature in the future. For local reordering, it may not be straightforward, as we have to work on both multiple-row height cells and single-row height cells together, making the enumeration of all permutations complicated.

VI. EXPERIMENTAL RESULTS

ABCDPlace was implemented with C++/CUDA for GPU and C++/OpenMP for multi-threaded CPU, respectively. The framework was validated under ISPD 2005 and 2015 contest benchmarks [47], [48] and a set of benchmarks from industry. We adopt sequential detailed placement engines in NTUplace3 [6] and NTUplace4dr [44] for comparison. The runtime environments for the three sets of benchmarks are slightly different, which are shown at the bottom of Table II, Table III, and Table IV, respectively. While the proposed parallel placement algorithms can be arbitrarily combined according to the real applications, we fixed the detailed placement flow in the experiment as the following sequence: local reordering, independent set matching, global swap, and local reordering to search for different solution spaces. All the runtime values reported in this section are wall-time for detailed placement excluding the file IO time, as in practice, all the data is already in the memory if running in the entire backend flow.

A. HPWL and Runtime Evaluation

We compare our parallel algorithms on both CPU and GPU with NTUplace3 [6] in terms of HPWL and runtime. The legalized global placement solutions are generated by an open-source placer DREAM-Place [28]. As Figure 1 indicates, NTUplace3 is very competitive in its efficiency. In Table II and Table III, we show the HPWL

and runtime for single-threaded, 10-thread, 20-thread, and GPU for ISPD 2005 contest benchmarks and industrial benchmarks. The file IO time is shown in separate columns for reference. As the file IO of ABCDPlace has a sequential implementation, we only show the time for single thread to save space. Multithreading has similar values. The GPU version has longer file IO time than the CPU versions, as we need to first read data from disk to CPU memory and then copy to GPU global memory. If we run a full placement flow, including global placement, legalization, and detailed placement, we can initialize all data in the GPU global memory. Thus, this is not a mandatory overhead. As mentioned in Section IV, we choose different algorithms for the maximal independent set and the partitioning steps in independent set matching to maximize the efficiency, so the wirelength results are slightly different, but they are almost the same on average.

On ISPD 2005 benchmarks, with a single thread, ABCDPlace demonstrates competitive runtime compared with NTUplace3, while ABCDPlace can achieve more than $2\times$ speedup with 20 threads, and more than $10\times$ speedup with GPU. On large industrial benchmarks, the speedup from multithreading is more than $4\times$ and that from GPU is more than $16\times$ on average. The difference in the speedup mainly comes from discrepant experimental environment for the two benchmarks and design sizes. Figure 13 plots the speedup over our single thread implementation versus the design sizes from 200K to 10M. Generally speaking, the speedup increases with the design sizes and saturates at 1M to 2M, especially on GPU. For million-size designs, the speedup values stay above $15\times$ for GPU, while the CPU speedup varies between $2\times -5\times$.

Another observation from the tables is that the speedup values are close between 10 and 20 threads, i.e., much less than the number of threads, indicating that the benefits from CPU parallelization saturate. With current implementations in the experiments, GPU acceleration provides more speedup than CPU multithreading, while the CPU parallelization may be further improved in the future.

B. Routability Evaluation

Although ABCDPlace does not explicitly consider routability so far, we perform experiments on ISPD 2015 contest benchmarks [48] to see whether it leads to significant overhead in congestion. The original objective of the contest is detailed-routability-driven placement and NTUplace4dr was the winner [44]. We obtained the binary from the NTUplace4dr team and conducted experiments with the legalized global placement solutions dumped by NTUplace4dr as the input. The results are shown in Table IV. As the original industrial evaluation platform for detailed routability is no longer available, we report the "top5 overflow", i.e., the average of the global routing overflow in the top 5% congested routing grids, evaluated from the NCTUgr global router [49] integrated in NTUplace4dr. It can be seen that besides improving the HPWL, ABCDPlace even slightly reduces the global routing overflow by 2.9%, which is even better than NTUplace4dr. This indicates that our algorithms do not harm the routability too much in these benchmarks. However, it also needs to be noted that although our global routing overflow is less than NTUplace4dr, it does not mean we can achieve better detailed routability. As NTUplace4dr spends a lot of efforts to optimize the detailed routing congestion issues, e.g., DRC errors, we expect ABCDPlace to have more DRC errors if detailed routing is performed.

For runtime comparison, we report the detailed placement runtime along with the file IO time for reference. As NTUplace4dr can run the entire placement flow and report the runtime in each stage, the file IO time is not retrieved and only the core detailed placement time

TABLE I: Summary of Major Differences in CPU and GPU Implementations

Algorithm		Multi-threaded CPU	GPU
Concurrent Independent Set Matching	Partitioning	Sequential spiral search	Parallel K-Means clustering
	Batch LAP	One thread for each LAP instance	One thread block for each LAP instance
Concurrent Global Swap	`CollectCands`	One thread for candidate collection of one cell	One thread block for candidate collection of one cell at one bin
	`CalcSwapCosts`	One thread for swap candidates of one cell	One thread for each swap candidate
	`FindBestCand`		2D parallel reduction for a matrix of candidates (batch size $\times$ max candidates per cell)
Concurrent Local Reordering	Parallelization granularity	One thread for each row in each independent row group	One thread for one permutation of one sliding window in a row in each independent row group

is reported. Meanwhile, due to the detailed-routability optimization in NTUplace4dr, it is slower than ABCDPlace even with a single thread. It is not very meaningful to compare placers with different objectives. We focus on the speedup over single-threaded CPU from multithreading and GPU. The average speedup is around $5\times$ with 20 threads and $6.8\times$ with GPU for all designs. However, for large designs, e.g., the `superblue` series, the speedup from GPU over a single thread can go to $25\times$ and beyond. In Figure 13, the speedup curve for GPU climbs up quickly with the increase of design sizes.

C. Runtime Breakdown

Figure 14 examines the runtime breakdown of NTUplace3 (representing sequential implementation), ABCDPlace with 20 threads and with GPU on ISPD 2005 benchmarks. NTUplace3 runs the local reordering and independent set matching steps twice and the runtime breakdown is around half and half. ABCDPlace also runs four steps, i.e., local reordering twice, independent set matching and global swap once. The runtime breakdown maps for CPU and GPU are similar. On CPU, independent set matching takes the largest portion, while on GPU, local reordering is most time-consuming. On ISPD 2015 benchmarks, the runtime distribution is different, as shown in Figure 15, where independent set matching takes the largest portion of the runtime for both CPU and GPU.

We also plot the speedup of each individual step from multi-threaded and GPUs over single-threaded execution, as shown in Figure 14d. Here we use the single-threaded version of the proposed concurrent algorithms as the baseline for fair comparison. With 10 and 20 threads, we can achieve around $5\times$ speedup for local reordering and global swap, as well as $3\times$ speedup for independent set matching. With GPU, the speedup can reach over $32\times$ for local reordering, $22\times$ for global swap, and $19\times$ for independent set matching.

Figure 16, 17, and 18 draw the runtime breakdown of each internal step for concurrent independent set matching, global swap, and local reordering, respectively.

- **Concurrent independent set matching**. The breakdown maps have different flavors between multi-threaded CPU and GPU implementation. Most of the efforts are spent on partitioning and maximal independent set for CPU, while for GPU, LAP and partitioning take the largest portions.
- **Concurrent global swap**. The breakdown maps for both multi-threaded CPU and GPU are similar. One may note that `ApplyCand` takes quite some portion of the runtime, because that is the only step that has to run sequentially.
- **Concurrent local reordering**. There are only two steps for this algorithm: an initialization step to compute the independent rows and an iterative enumeration step. The initialization step is done sequentially on CPU, while the enumeration step runs for two iterations in Figure 18. With multi-threaded CPU, enumeration takes over 99% of the runtime, while GPU implementation

significantly accelerates this part such that the portion of initialization becomes not negligible.

These breakdown maps for the concurrent algorithms show the effectiveness of our acceleration techniques to speedup the critical portions of the computation and achieve more balanced runtime distribution of each step.

D. Clarification to the Combination of Placement Techniques

In all the experiments, we apply the placement techniques in the following sequence: local reordering, independent set matching, global swap, and local reordering. It needs to be clarified that this combination is empirically determined according to the experiments and these techniques can be arbitrarily combined. Users are encouraged to customize the combination according to their benchmarks. Intuitively, it is better to interleave different techniques, as they explore different solution spaces. Thus, we go through local reordering, independent set matching, and global swap each once. Then, we find the wirelength improvement mostly saturates after applying another round of local reordering, so we choose the current combination for the experiments.

VII. CONCLUSION

We present *ABCDPlace*, an open-source batch-based acceleration of detailed placement on multi-threaded CPUs and GPUs. We propose efficient parallel algorithms for classic sequential detailed placement techniques based on batch execution. The placer can achieve over $10\times$ and $16\times$ speedup with GPUs on ISPD 2005 contest benchmarks and industrial benchmarks, respectively, without quality degradation. The multi-threaded CPU version also achieves around $2\times-5\times$ speedup. For a 10-million-cell design, our placer is able to finish within one minute on GPU, while it takes almost half an hour for a sequential implementation like NTUplace3. Experiments on ISPD 2015 benchmarks also show that the placer has minimal overhead in global routing congestion, even though routability is not explicitly considered. We believe the parallelization strategies can shed lights to accelerating other sequential design automation algorithms for fast design closure.

In the future, there are many perspectives to further improve ABCDPlace.

- Incorporate holistic optimization objectives such as routability, timing, and multiple-row height cells.
- Better parallelization. Current speedup for CPU is still limited and there is room to improve. We also consider to explore other parallelization strategies such as diagonal partitioning.
- Incorporate more detailed placement techniques such as row-based placement algorithms.

As an open-source project, ABCDPlace can provide an initial development platform for efficient and effective placement engines.

TABLE II: Comparison of runtime (in seconds) and HPWL with NTUplace3 [6] on ISPD 2005 contest benchmarks. "1T", "10T", and "20T" denote single, 10, and 20 threads, respectively. "RT" denotes the core detailed placement runtime and "IO" denotes the file IO time.

Design	#cells	#nets	Initial HPWL	NTUplace3 Single thread			ABCDPlace 1T			10T		20T		GPU		
				HPWL	RT	IO	HPWL	RT	IO	HPWL	RT	HPWL	RT	HPWL	RT	IO
adaptec1	211K	221K	74.35	73.28	25	3	73.23	38	6	73.21	11	73.21	10	73.21	3	13
adaptec2	254K	266K	83.22	82.14	32	4	82.03	44	7	82.03	15	82.03	18	82.05	4	14
adaptec3	451K	467K	199.01	193.98	59	7	193.32	96	12	193.21	24	193.21	22	193.47	7	21
adaptec4	495K	516K	178.28	174.40	68	7	174.41	94	13	174.41	25	174.41	25	174.49	6	23
bigblue1	278K	284K	90.18	89.44	35	4	89.46	47	8	89.45	15	89.45	14	89.43	4	15
bigblue2	535K	577K	139.46	136.76	95	8	136.92	97	15	136.92	24	136.92	23	137.00	6	25
bigblue3	1093K	1123K	310.91	303.98	148	15	304.14	196	28	304.17	59	304.17	57	304.46	10	43
bigblue4	2169K	2230K	751.08	743.75	354	34	744.46	369	61	744.42	91	744.42	80	744.35	16	88
ratio				1.017	1.000	1.000	1.000	1.330		1.000	0.380	1.000	0.369	1.000	0.091	

The CPU results for the ISPD 2005 benchmarks were collected from a Linux server with a 20-core Intel Xeon E5-2650 v3 @ 2.3GHz. The GPU results were collected from a Linux server with a 15-core Intel Xeon Silver 4110 CPU @ 2.1GHz CPU and an NVIDIA Tesla V100 GPU.

TABLE III: Comparison of runtime (in seconds) and HPWL with NTUplace3 [6] on industrial benchmarks. "RT" denotes the core detailed placement runtime and "IO" denotes the file IO time.

Design	#cells	#nets	Initial HPWL	NTUplace3 Single thread			ABCDPlace 1T			10T		20T		GPU		
				HPWL	RT	IO	HPWL	RT	IO	HPWL	RT	HPWL	RT	HPWL	RT	IO
design1	1329K	1389K	346.23	340.96	194	35	341.04	236	40	341.03	59	341.03	42	341.00	14	42
design2	1300K	1355K	281.45	275.79	203	33	275.63	232	41	275.64	56	275.64	41	275.56	13	43
design3	2246K	2276K	531.93	523.06	332	58	522.93	384	66	522.97	95	522.97	78	522.98	19	69
design4	1512K	1528K	459.49	454.14	233	41	453.97	292	47	453.99	65	453.99	51	453.91	15	49
design5	1306K	1364K	294.05	288.38	203	34	288.47	236	39	288.49	55	288.49	41	288.45	13	43
design6	10504K	10747K	2348.81	2346.33	1565	331	2348.64	1391	331	2348.65	442	2348.65	349	2348.34	58	350
ratio				1.014	1.000	1.000	1.000	1.137		1.000	0.283	1.000	0.215	1.000	0.059	

The results for the industrial benchmarks were collected from a Linux server with a 20-core Intel E5-2698 v4 CPU @ 2.2GHz CPU and an NVIDIA Tesla V100 GPU.

TABLE IV: Comparison of runtime (in seconds), HPWL, and congestion with NTUplace4dr [44] on ISPD 2015 contest benchmarks. "1T"and "20T" denote single and 20 threads, respectively. "RT" denotes the core detailed placement runtime and "IO" denotes the file IO time.

	#cells	#nets	Initial HPWL	Initial Top5 Overflow	NTUplace4dr			ABCDPlace 1T				ABCDPlace 20T			ABCDPlace GPU			
					HPWL	Top5 Overflow	RT	HPWL	Top5 Overflow	RT	IO	HPWL	Top5 Overflow	RT	HPWL	Top5 Overflow	RT	IO
mgc_des_perf_1	113K	113K	1.22	64.97	1.19	63.89	76	1.16	63.17	27	3	1.16	63.32	6	1.16	62.58	5	9
mgc_des_perf_a	108K	115K	2.47	74.45	2.44	73.14	91	2.37	71.53	45	3	2.37	71.54	9	2.37	71.59	5	9
mgc_des_perf_b	113K	113K	2.02	72.44	2.00	71.71	80	1.95	70.68	42	3	1.95	70.22	9	1.95	70.51	5	10
mgc_edit_dist_a	127K	134K	5.03	93.71	4.91	92.95	80	4.60	92.12	61	3	4.60	91.97	12	4.62	92.13	6	10
mgc_fft_1	32K	33K	0.46	61.59	0.46	61.87	22	0.44	59.08	3	1	0.44	59.39	1	0.44	59.36	2	8
mgc_fft_2	32K	33K	0.48	55.20	0.48	55.10	28	0.48	54.76	2	1	0.48	55.00	1	0.48	54.78	1	6
mgc_fft_a	31K	32K	0.65	36.91	0.64	36.07	34	0.63	35.53	3	1	0.63	35.54	2	0.63	35.59	2	8
mgc_fft_b	31K	32K	0.86	53.84	0.85	53.19	26	0.84	52.87	3	1	0.84	52.80	1	0.84	52.86	1	7
mgc_matrix_mult_1	155K	159K	2.30	73.71	2.27	73.43	61	2.21	72.85	27	4	2.21	72.53	6	2.21	72.77	3	10
mgc_matrix_mult_2	155K	159K	2.28	73.87	2.25	73.44	65	2.21	72.66	26	4	2.21	72.24	7	2.21	72.37	3	10
mgc_matrix_mult_a	150K	154K	3.37	49.04	3.33	48.62	87	3.32	48.49	24	4	3.32	48.48	5	3.32	48.51	3	11
mgc_matrix_mult_b	146K	152K	3.67	49.80	3.63	49.29	69	3.61	48.81	30	4	3.61	48.79	7	3.61	48.88	5	11
mgc_matrix_mult_c	146K	152K	3.51	48.84	3.47	48.22	69	3.46	48.09	27	4	3.46	48.03	6	3.46	48.11	5	11
mgc_pci_bridge32_a	30K	34K	0.47	41.74	0.47	41.52	25	0.47	40.49	5	1	0.47	40.75	2	0.46	40.28	4	7
mgc_pci_bridge32_b	29K	33K	0.70	35.90	0.69	35.38	23	0.67	34.40	7	1	0.67	34.48	3	0.68	34.49	5	7
mgc_superblue11_a	926K	936K	39.67	58.69	38.99	57.64	4297	38.41	57.04	874	30	38.41	57.08	102	38.42	57.05	31	40
mgc_superblue12	1287K	1293K	35.52	79.28	34.76	78.84	1805	34.09	78.08	700	41	34.10	78.12	90	34.10	78.08	24	46
mgc_superblue14	612K	620K	25.41	70.31	24.85	67.76	751	24.19	66.30	933	20	24.20	66.52	115	24.22	66.53	35	26
mgc_superblue16_a	680K	697K	31.24	97.45	30.34	94.24	746	29.17	90.24	831	22	29.16	90.20	86	29.25	90.55	30	27
mgc_superblue19	506K	512K	17.45	65.08	17.13	64.55	967	16.97	63.80	287	16	16.97	63.88	41	16.99	64.01	14	22
ratio				1.015	1.013	1.000	1.000	1.000	0.979	0.984	0.412		0.979	0.985	0.084	0.979	0.984	0.061	

The CPU results for the ISPD 2015 benchmarks were collected from a Linux server with a 20-core Intel Xeon E5-2650 v3 @ 2.3GHz. The GPU results were collected from a Linux server with a 14-core Intel Xeon E5-2690 v4 @ 2.6GHz and an NVIDIA Tesla V100 GPU.
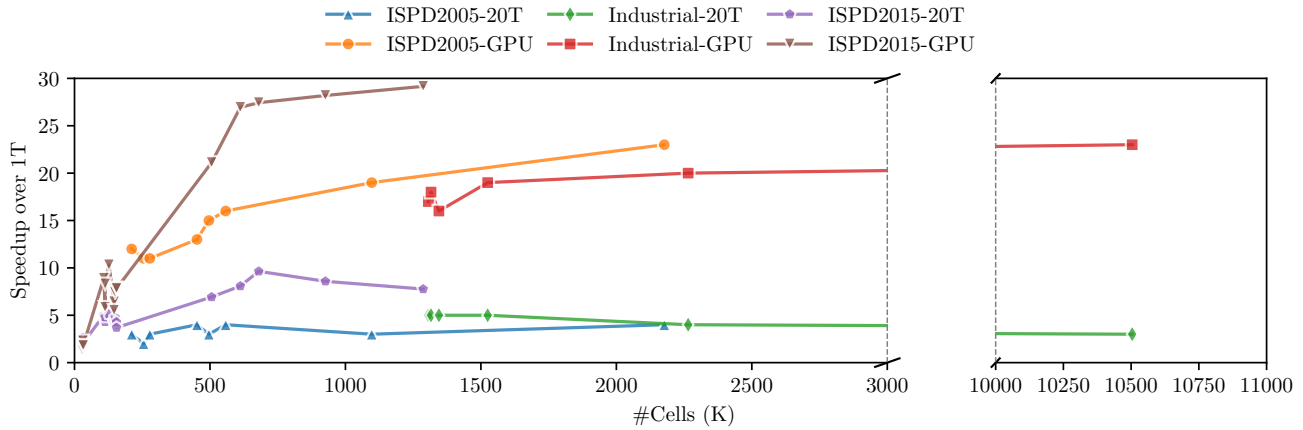
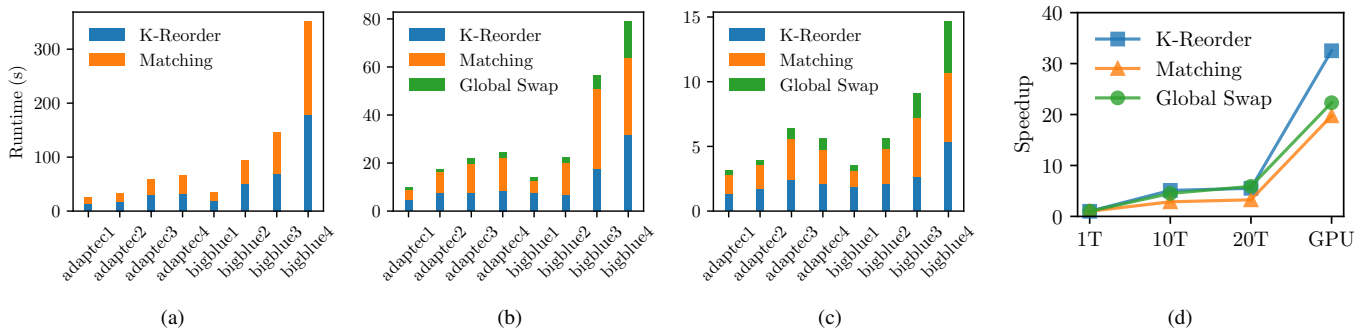Fig. 13: The trend of speedup over single thread with design sizes. "T" stands for threads on CPU.

Fig. 14: Runtime breakdown of ISPD 2005 benchmarks for (a) single-threaded NTUplace3, (b) ABCDPlace with 20 threads and (c) with GPU. (d) Speedup of each step with number of threads on bigblue4.
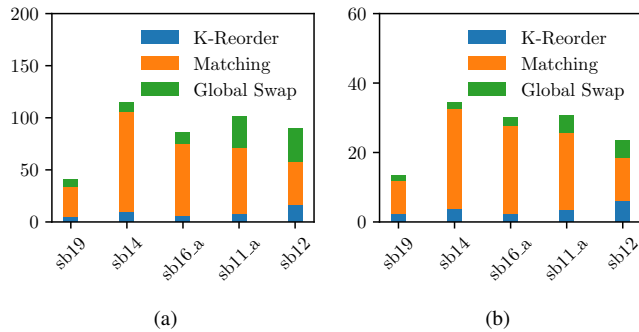

Fig. 15: Runtime breakdown of ISPD 2015 benchmarks for (a) ABCDPlace with 20 threads and (b) with GPU. Only mgc_superblue series (abbreviated as "sb") are plotted as the other benchmarks are too small to be representative.
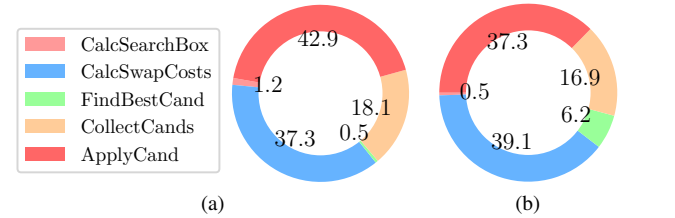
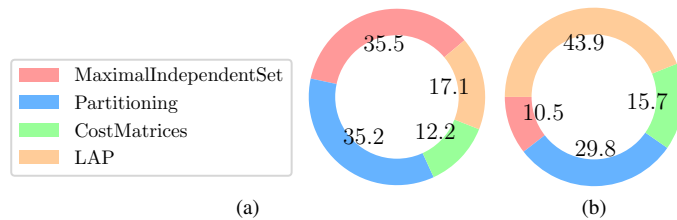Fig. 17: Runtime breakdown for global swap on bigblue4: (a) 20 threads; (b) GPU.

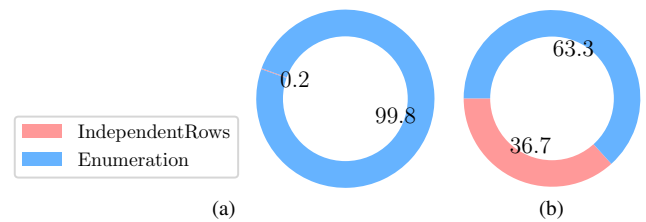Fig. 16: Runtime breakdown for independent set matching on bigblue4: (a) 20 threads; (b) GPU.

Fig. 18: Runtime breakdown for local reordering on bigblue4: (a) 20 threads; (b) GPU.

11

REFERENCES

[1] K. Shahookar and P. Mazumder, "Vlsi cell placement techniques," *ACM Computing Surveys (CSUR)*, vol. 23, no. 2, pp. 143–220, 1991.

[2] M. Pan, N. Viswanathan, and C. Chu, "An efficient and effective detailed placement algorithm," in *Proc. ICCAD*, 2005, pp. 48–55.

[3] Z.-W. Jiang, H.-C. Chen, T.-C. Chen, and Y.-W. Chang, "Challenges and solutions in modern vlsi placement," in *2007 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 2007, pp. 1–5.

[4] S. Popovych, H.-H. Lai, C.-M. Wang, Y.-L. Li, W.-H. Liu, and T.-C. Wang, "Density-aware detailed placement with instant legalization," in *Proc. DAC*, 2014, pp. 122:1–122:6.

[5] M.-K. Hsu, Y.-F. Chen, C.-C. Huang, S. Chou, T.-H. Lin, T.-C. Chen, and Y.-W. Chang, "NTUplace4h: A novel routability-driven placement algorithm for hierarchical mixed-size circuit designs," *IEEE TCAD*, vol. 33, no. 12, pp. 1914–1927, 2014.

[6] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE TCAD*, vol. 27, no. 7, pp. 1228–1240, 2008.

[7] W.-K. Chow, J. Kuang, X. He, W. Cai, and E. F. Young, "Cell density-driven detailed placement with displacement constraint," in *Proceedings of the 2014 on International symposium on physical design*. ACM, 2014, pp. 3–10.

[8] A. B. Kahng, I. L. Markov, and S. Reda, "On legalization of row-based placements," in *Proc. GLSVLSI*, 2004, pp. 214–219.

[9] J. Chen, P. Yang, X. Li, W. Zhu, and Y.-W. Chang, "Mixed-cell-height placement with complex minimum-implant-area constraints," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 66.

[10] I. L. Markov, J. Hu, and M. Kim, "Progress and challenges in VLSI placement research," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 1985–2003, 2015. [Online]. Available: https://doi.org/10.1109/JPROC.2015.2478963

[11] N. Viswanathan and C.-N. 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*, vol. 24, no. 5, pp. 722–733, 2005.

[12] W.-K. Chow, J. Kuang, X. He, W. Cai, and E. F. Y. Young, "Cell density-driven detailed placement with displacement constraint," in *Proc. ISPD*, 2014, pp. 3–10.

[13] G. Wu and C. Chu, "Detailed placement algorithm for VLSI design with double-row height standard cells," *IEEE TCAD*, vol. 35, no. 9, pp. 1569–1573, 2016.

[14] K. Han, A. B. Kahng, and H. Lee, "Scalable detailed placement legalization for complex sub-14nm constraints," in *Proc. ICCAD*, 2015, pp. 867–873.

[15] T. Lin and C. Chu, "TPL-aware displacement-driven detailed placement refinement with coloring constraints," in *Proc. ISPD*, 2015, pp. 75–80.

[16] Y. Lin, B. Yu, B. Xu, and D. Z. Pan, "Triple patterning aware detailed placement toward zero cross-row middle-of-line conflict," *IEEE TCAD*, vol. 36, no. 7, pp. 1140–1152, 2017.

[17] J. Chen, P. Yang, X. Li, W. Zhu, and Y.-W. Chang, "Mixed-cell-height placement with complex minimum-implant-area constraints," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 66.

[18] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE TCAD*, 2018.

[19] Z. Zhu, J. Chen, Z. Peng, W. Zhu, and Y.-W. Chang, "Generalized augmented lagrangian and its applications to vlsi global placement," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[20] W. Zhu, Z. Huang, J. Chen, and Y.-W. Chang, "Analytical solution of poisson's equation and its application to vlsi global placement," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[21] F.-K. Sun and Y.-W. Chang, "Big: A bivariate gradient-based wirelength model for analytical circuit placement," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 118.

[22] J. A. Chandy and P. Banerjee, "Parallel simulated annealing strategies for vlsi cell placement," in *Proceedings of 9th International Conference on VLSI Design*, Jan 1996, pp. 37–42.

[23] A. Choong, R. Beidas, and J. Zhu, "Parallelizing simulated annealing-based placement using gpgpu," in *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 2010, pp. 31–34.

[24] C. Fobel, G. Grewal, and D. Stacey, "A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and gpu architectures," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–8.

[25] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *Proc. ICCAD*. ACM, 2009, pp. 681–688.

[26] T. Lin, C. Chu, and G. Wu, "Polar 3.0: An ultrafast global placement engine," in *Proc. ICCAD*. IEEE, 2015, pp. 520–527.

[27] W. Li, M. Li, J. Wang, and D. Z. Pan, "UTPlaceF 3.0: A parallelization framework for modern fpga global placement," in *Proc. ICCAD*. IEEE, 2017, pp. 922–928.

[28] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Pan, "DREAM-Place: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," in *Proc. DAC*, 2019.

[29] S. Dhar and D. Z. Pan, "GDP: GPU accelerated detailed placement," in *Proc. HPEC*, Sept 2018.

[30] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. Wong, "Cpp-taskflow: Fast parallel programming with task dependency graphs," *Proc. IPDPS*, 2019.

[31] Y.-S. Lu and K. Pingali, *Can Parallel Programming Revolutionize EDA Tools?* Cham: Springer International Publishing, 2018, pp. 21–41. [Online]. Available: https://doi.org/10.1007/978-3-319-67295-3_2

[32] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: Gpu graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, p. 3, 2017.

[33] Y. Lin, B. Yu, X. Xu, J.-R. Gao, N. Viswanathan, W.-H. Liu, Z. Li, C. J. Alpert, and D. Z. Pan, "Mrdp: Multiple-row detailed placement of heterogeneous-sized cells for advanced nodes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 6, pp. 1237–1250, 2017.

[34] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," in *Proc. ASPDAC*, 2007, pp. 135–140.

[35] T.-C. Chen, T.-C. Hsu, Z.-W. Jiang, and Y.-W. Chang, "NTUplace: a ratio partitioning based placement algorithm for large-scale mixed-size designs," in *Proc. ISPD*, 2005, pp. 236–238.

[36] F. Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph," *SIAM Journal on Computing (SICOMP)*, vol. 1, no. 2, pp. 180–187, 1972.

[37] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy sequential maximal independent set and matching are parallel on average," *CoRR*, vol. abs/1202.3205, 2012. [Online]. Available: http://arxiv.org/abs/1202.3205

[38] D. P. Bertsekas, "A new algorithm for the assignment problem," *Mathematical Programming*, vol. 21, no. 1, pp. 152–171, 1981.

[39] M. M. Zavlanos, L. Spesivtsev, and G. J. Pappas, "A distributed auction algorithm for the assignment problem," in *2008 47th IEEE Conference on Decision and Control*. IEEE, 2008, pp. 1212–1217.

[40] "Munkres-CPP," https://github.com/saebyn/munkres-cpp.

[41] "LEMON," http://lemon.cs.elte.hu/trac/lemon.

[42] V. Kumar, *Introduction to parallel computing*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[43] J. Jaja, *An introduction to parallel algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[44] C.-C. Huang, H.-Y. Lee, B.-Q. Lin, S.-W. Yang, C.-H. Chang, S.-T. Chen, Y.-W. Chang, T.-C. Chen, and I. Bustany, "Ntuplace4dr: a detailed-routing-driven placer for mixed-size circuit designs with technology and region constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 3, pp. 669–681, 2017.

[45] J. Jung, G.-J. Nam, L. N. Reddy, I. H.-R. Jiang, and Y. Shin, "Owaru: Free space-aware timing-driven incremental placement with critical path smoothing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1825–1838, 2017.

[46] J. A. S. Jesuthasan, "Incremental timing-driven placement with displacement constraint," Master's thesis, University of Waterloo, 2015.

[47] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "The ispd2005 placement contest and benchmark suite," in *Proc. ISPD*. ACM, 2005, pp. 216–220.

[48] I. S. Bustany, D. Chinnery, J. R. Shinnerl, and V. Yutsis, "ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement," in *Proc. ISPD*, 2015, pp. 157–164.

[49] K.-R. Dai, W.-H. Liu, and Y.-L. Li, "NCTU-GR: efficient simulated evolution-based rerouting and congestion-relaxed layer assignment on 3-D global routing," *IEEE TVLSI*, vol. 20, no. 3, pp. 459–472, 2012.

Brucek Khailany (M'00–SM'13) received the the Ph.D. degree from Stanford University in Stanford, CA in 2003 and the B.S.E. degree from the University of Michigan in Ann Arbor, MI in 1997, in electrical engineering. He joined NVIDIA in 2009 and is currently the Director of the ASIC and VLSI Research group. He leads research into innovative design methodologies for integrated circuit (IC) development, machine learning (ML) and GPU-assisted electronic design automation (EDA) algorithms, and energy-efficient ML accelerators. Over 10 years at NVIDIA, he has contributed to many projects in research and product groups spanning computer architecture and VLSI design. Previously, from 2004-2009, he was a Co-Founder and Principal Architect at Stream Processors, Inc (SPI) where he led research and development activities related to parallel processor architectures.

Yibo Lin (S'16–M'19) received the B.S. degree in microelectronics from Shanghai Jiaotong University in 2013, and his Ph.D. degree from the Electrical and Computer Engineering Department of the University of Texas at Austin in 2018. He is current an assistant professor in the Computer Science Department associated with the Center for Energy-Efficient Computing and Applications at Peking University, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security. He has received 3 Best Paper Awards at premier venues (DAC 2019, VLSI Integration 2018, and SPIE 2016). He has also served in the Technical Program Committees of many major conferences, including ICCAD, ICCD, ISPD, and DAC.
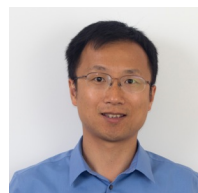
Wuxi Li (S'18–M'19) received the B.S. degree in microelectronics from Shanghai Jiao Tong University, Shanghai, China, in 2013., the M.S. and Ph.D. degrees in computer engineering from the University of Texas at Austin, Austin, TX, in 2015 and 2019, respectively. He is currently a Staff Software Engineer in the Vivado Implementation Team at Xilinx, San Jose, CA, where he is primarily working on the physical synthesis field.

Dr. Li has received the Best Paper Award at DAC 2019, the Silver Medal in ACM Student Research Contest at ICCAD 2018, and the 1st-place awards in the FPGA placement contests of ISPD 2016 and 2017.

David Z. Pan (S'97—M'00—SM'06—F'14) received his B.S. degree from Peking University, and his M.S. and Ph.D. degrees from University of California, Los Angeles (UCLA). From 2000 to 2003, he was a Research Staff Member with IBM T. J. Watson Research Center. He is currently Engineering Foundation Professor at the Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA. His research interests include cross-layer nanometer IC design for manufacturability, reliability, security, machine learning and hardware acceleration, design/CAD for analog/mixed signal designs and emerging technologies. He has published over 360 journal articles and refereed conference papers, and is the holder of 8 U.S. patents.

He has served as a Senior Associate Editor for ACM Transactions on Design Automation of Electronic Systems (TODAES), an Associate Editor for IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD), IEEE Transactions on Very Large Scale Integration Systems (TVLSI), IEEE Transactions on Circuits and Systems PART I (TCAS-I), IEEE Transactions on Circuits and Systems PART II (TCAS-II), IEEE Design & Test, Science China Information Sciences, Journal of Computer Science and Technology, IEEE CAS Society Newsletter, etc. He has served in the Executive and Program Committees of many major conferences, including DAC, ICCAD, ASPDAC, and ISPD. He is the ASPDAC 2017 Program Chair, ICCAD 2018 Program Chair, DAC 2014 Tutorial Chair, and ISPD 2008 General Chair

He has received a number of prestigious awards for his research contributions, including the SRC Technical Excellence Award in 2013, DAC Top 10 Author in Fifth Decade, DAC Prolific Author Award, ASP-DAC Frequently Cited Author Award, 18 Best Paper Awards at premier venues (ASPDAC 2020, DAC 2019, GLSVLSI 2018, VLSI Integration 2018, HOST 2017, SPIE 2016, ISPD 2014, ICCAD 2013, ASPDAC 2012, ISPD 2011, IBM Research 2010 Pat Goldberg Memorial Best Paper Award, ASPDAC 2010, DATE 2009, ICICDT 2009, SRC Techcon in 1998, 2007, 2012 and 2015) and 15 additional Best Paper Award finalists, Communications of the ACM Research Highlights (2014), UT Austin RAISE Faculty Excellence Award (2014), and many international CAD contest awards, among others. He is a Fellow of IEEE and SPIE.
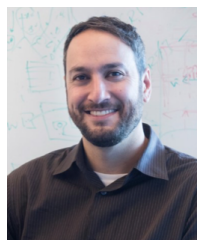
Jiaqi Gu received the B.E. degree in Microelectronic Science and Engineering from Fudan University, Shanghai, China in 2018. He is currently a postgraduate student studying for his Ph.D. degree in the Department of Electrical and Computer Engineering, The University of Texas at Austin. His current research interests include machine learning, optical neuromorphic computing for AI acceleration, and GPU acceleration for VLSI placement.

Haoxing Ren (M'00–SM'09) received his B.S/M.S. degrees in Electrical Engineering from Shanghai Jiao Tong University, his M.S. degree in Computer Engineering from Rensselaer Polytechnic Institute, and his PhD degree in Computer Engineering from University of Texas at Austin. From 2000 to 2006, he was a software engineer with IBM Microelectronics. From 2007 to 2015, he was a Research Staff Member with IBM T. J. Watson Research Center. From 2015 to 2016, he was a technical executive with SuZhou PowerCore Technology. He is currently a Principal Research Scientist at NVIDIA. His research interests are machine learning applications in design automation and GPU accelerated EDA. He received many IBM technical achievement rewards including the IBM Corporate Award for his work on improving microprocessor design productivity. He holds over twenty patents and co-authored more than 40 papers including several book chapters in physical design and logic synthesis. He has received the Best Paper Awards at ISPD'13 and DAC'19.