Ultrafast CPU/GPU Kernels for Density Accumulation in Placement

Zizheng Guo*
CECA, CS Department
Peking University
Beijing, China
gzz@pku.edu.cn

Jing Mai*
CECA, CS Department
Peking University
Beijing, China
magic3007@pku.edu.cn

Yibo Lin†
CECA, CS Department
Peking University
Beijing, China
yibolin@pku.edu.cn

***Abstract*—Density accumulation is a widely-used primitive operation in physical design, especially for placement. Iterative invocation in the optimization flow makes it one of the runtime bottlenecks. Accelerating density accumulation is challenging due to data dependency and workload imbalance. In this paper, we propose efficient CPU/GPU kernels for density accumulation by decomposing the problem into two phases: constant-time density collection for each instance and a linear-time prefix sum. We develop CPU and GPU dedicated implementations, and demonstrate promising efficiency benefits on tasks from large-scale placement problems.**

I. Introduction

Density accumulation is a widely-used operation in physical design, such as placement and routing. It can be used for characterizing the density distributions of rectangular shapes on an $M \times N$ grid system. There are two typical variations of density accumulation: the *forward* accumulation to compute the density map from a set of rectangular shapes, and the *backward* one to accumulate the weights from a density map to the shapes. As a primitive operation, it can take a significant portion of the optimization time due to iterative calls.

Density accumulation is a required kernel in many placers such as POLAR [1], NTUplace series [2], [3], and ePlace series [4], [5]. For example, in nonlinear placement, it is used in each gradient descent iteration of the following optimization problem [2]–[9],

$$\min_{\mathbf{x},\mathbf{y}} WL(\mathbf{x},\mathbf{y}) + \lambda \mathcal{D}(\mathbf{x},\mathbf{y}), \tag{1}$$

where $\mathbf{x}, \mathbf{y}$ are cell locations, the first term $\sum WL(\cdot)$ models the total wirelength of nets, and the second term describes the density penalty to avoid overlaps between cells. In each iteration, computing the penalty $\mathcal{D}(\cdot)$ needs the density map of cells, i.e., the forward density accumulation; computing $\frac{\partial \mathcal{D}}{\partial \mathbf{x}}, \frac{\partial \mathcal{D}}{\partial \mathbf{y}}$ needs the backward density accumulation to obtain the density gradient of each cell. Our profiling of a recent open-source GPU-accelerated placer [8] indicates around 60% of the runtime taken by density computation for each gradient descent iteration on million-cell designs [10], most of which lies within density accumulation. As nonlinear placement typically requires hundreds or thousands of iterations to converge, the performance of density accumulation plays a critical role in the overall efficiency.

Density accumulation is also a primitive operation in routability estimation models like RISA and RUDY [11], [12]. These models divide the layout into $M \times N$ grids and distribute the routing demands of each net into the grids covered by net bounding boxes, which is essentially a forward density accumulation. As net bounding box computation is fully parallelized, over 95% of the runtime is taken by density accumulation on GPU for million-cell designs [8], [13]. In routability-driven placement algorithms based on cell inflation, forward accumulation is used in routability estimation and backward accumulation is involved in computing the inflation ratios for each cell [5], [8].

While density accumulation is widely-used in various physical design stages, its performance is often limited and becomes the runtime bottlenecks in iterative optimization. There are several challenges in accelerating this operation with a manycore system or a GPU platform. 1) Its primitive task of updating one rectangular shape is too small to afford the large threading overhead. 2) The workload can be very imbalanced due to heterogeneous-sized shapes. 3) Writing to the resulting density map in the forward density accumulation needs to avoid data race between threads.

Existing work has investigated parallelization of forward density accumulation on both CPU [9], [14] and GPU [8], [9]. Lin *et al* [9] accelerate forward accumulation by exploring efficient CPU atomic primitives, and reproducible GPUs kernel using fixed-point numbers. They also overlap data transfers with computations considering a heterogenous CPU-GPU scenario. Gessler *et al* [14] accelerate forward accumulation on CPU and achieve $3.2\times$ speedup with 12 threads by allocating thread-local copies of data to avoid synchronization overhead at the cost of more memory. Lin *et al* tried to assign multiple threads for updating each shape on GPU [8]. They achieve $\sim 2\times$ speedup over the implementation using one thread per shape. All these experiments are based on a standard cell placement flow, where the size of each cell is comparable to the grid size. However, the performance of the aforementioned techniques degrades at large shapes covering many grids, like the bounding boxes of nets in the routability modeling or large macros in mixed-sized placement problems, as the number of primitive operations is correlated to how many grids a shape covers.

To overcome the challenges, in this paper, we propose a generic and ultrafast algorithm for both variations of density accumulation, naturally supporting CPU and GPU platforms. We decompose the problem into two phases: a constant-time density collection phase for each instance and a linear-time prefix sum phase. In this way, we can achieve significant speedup over the existing implementations and the performance of our algorithm is insensitive to the heterogeneous-sized shapes. Experiments on ISPD 2005 & 2015 [10], [15] benchmarks demonstrate up to $22\times$ speedup on CPU and up to $64\times$ speedup on GPU compared with the state-of-the-art implementations [8], [14] on the same platforms, respectively.

The rest of the paper is organized as follows. Section II introduces the problem formulations. Section III explains the algorithms in detail. Section IV validates the algorithm with experimental results and Section V concludes the paper.

II. Preliminaries

Density accumulation deals with a set of rectangular instances V and an $M \times N$ grid system G. The instance here refers to rectangular objects like cells or nets in placement. We consider two variations in this work: *forward* and *backward* density accumulation, which are essentially inverse problems of each other. Figure 1 gives examples of the two variations. Forward accumulation computes the density map of instances on the grid system, and backward accumulation sums the weights on grids that overlap with an instance. To formally define the problems, we introduce the notations in Table I.

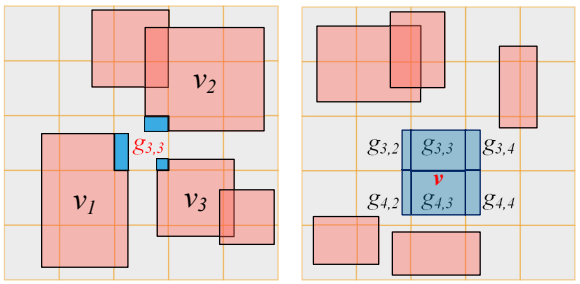
Problem 1 (Forward Density Accumulation)**.** *Given a set of instances V, an $M \times N$ grid system G, and the weight w_v^{inst} of each instance*

*Equal contribution, ordered by last names. †Corresponding author.

978-1-6654-3274-0/21/$31.00

TABLE I: Notations

Notation	Description
V	The set of instances
Box_v	The bounding box of the instance $v \in V$
$area_v^{inst}$	The area of the bounding box Box_v
w_v^{inst}	The weight of the instance $v \in V$
G	The set of grids in the $M \times N$ grid system
$g_{x,y}$	The grid indexed by (x, y), $x \in \{1, 2, \ldots, M\}, y \in \{1, 2, \ldots, N\}$
$w_{x,y}^{grid}$	The weight of the grid $g_{x,y} \in G$
$area_{x,y}^{grid}$	The area of the grid $g_{x,y}$



(a) (b)

Fig. 1: An illustration on forward and backward density accumulation. (a) Forward accumulation computes the density of grids. The density of $g_{3,3}$ is denoted as $\rho_{3,3}^{grid}$, which is computed from its intersections with 3 instances: v_1, v_2, and v_3. (b) Backward accumulation computes the density of instances. The density of instance v is denoted as ρ_v^{inst}, which is computed from its intersections with 6 grids: $g_{3,2}$, $g_{3,3}$,$g_{3,4}$, $g_{4,2}$, $g_{4,3}$ and $g_{4,4}$.

v, compute the density map ρ^{grid} on the grid system. Density $\rho_{x,y}^{grid}$ for each grid $g_{x,y} \in G$ is defined as follows:

$$\rho_{x,y}^{grid} = \sum_{v \in V} w_v^{inst} \times \frac{OA(Box_v, g_{x,y})}{area_{x,y}^{grid}}, \quad \forall g_{x,y} \in G, \tag{2}$$

where w_v^{inst} is the weight of instance v, $area_{x,y}^{grid}$ is the area of the grid $g_{x,y}$, and $OA(Box_v, g_{x,y})$ is the overlapping area between the bounding box of instance v and grid $g_{x,y}$.

Problem 2 (Backward Density Accumulation)**.** *Given a set of instances V, an $M \times N$ grid system G, and the weight $w_{x,y}^{grid}$ of each grid $g_{x,y}$, computes the density array ρ^{inst} for all instances. Density ρ_v^{inst} for each instance v is defined as follows:*

$$\rho_v^{inst} = \sum_{g_{x,y} \in G} w_{x,y}^{grid} \times \frac{OA(Box_v, g_{x,y})}{area_v^{inst}}, \forall v \in V, \tag{3}$$

where $w_{x,y}^{grid}$ is the weight of grid $g_{x,y}$, $area_v^{inst}$ is the area of instance v, and $OA(Box_v, g_{x,y})$ is the overlapping area between the bounding box of instance v and grid $g_{x,y}$.

Equation (2) takes the weights of instances and accumulates density for each grid, while Equation (3) takes the weights of grids and accumulates density for each instance. A typical approach to solve them is to enumerate the overlapping grids and collect results for each instance [8], [14]. To parallelize the computation, multiple threads are allocated to simultaneously loop through instances. However, heterogeneous-sized instances cause imbalanced workloads between threads, resulting in low performance and poor scalability of the algorithm.

III. Algorithms

We overcome the challenge of accelerating density accumulation problem by transforming forward/backward problems into equivalent forms, consisting of two phases: a linear-time two-dimensional (2D) prefix sum phase and a density collection phase whose time complexity is irrelevant to the sizes of instances.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \xrightarrow{\text{Sum}} P = \begin{pmatrix} 1 & 3 & 6 & 10 \\ 6 & 14 & 24 & 36 \\ 15 & 33 & 54 & 78 \\ 28 & 60 & 96 & 136 \end{pmatrix}$$

Fig. 2: An example of 2D prefix sum on a 4×4 matrix.

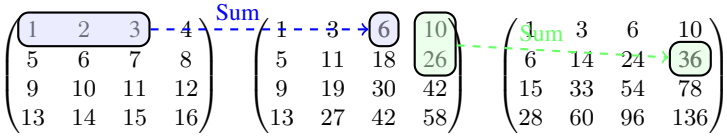


Fig. 3: Illustration of row sum followed by column sum. This is equivalent to a 2D prefix sum.

In this section, we first introduce the 2D prefix sum phase, which is used in solving both forward/backward problems. Then, we explain the details of the forward/backward algorithms. In the end, we illustrate the GPU-specific implementation.

A. 2D Prefix Sum

Both our forward and backward algorithms require to compute 2D prefix sum on a matrix A. The result matrix P has the same dimension as matrix A, with element $P_{i,j}$ equals to the sum of all values in A which are above it or on left of it. Figure 2 shows an example of 2D prefix sum on a 4×4 matrix. Each element in matrix P can be written as,

$$P_{i,j} = \sum_{x=1}^{i} \sum_{y=1}^{j} A_{x,y}, \tag{4}$$

where $i = 1, 2, \ldots, M, j = 1, 2, \ldots, N$, $A \in \mathcal{R}^{M \times N}$, and $P \in \mathcal{R}^{M \times N}$.

Figure 3 shows that 2D prefix sum can be computed by performing one-dimensional (1D) prefix sum along rows and then along columns. Based on this fact, we develop our algorithm for computing 2D prefix sum in Algorithm 1, which first computes sum along rows (lines 3-5) and then columns (lines 6-8). We can see that the time complexity of Algorithm 1 is $O(MN)$, which is linear in the size of the matrix.

B. Forward Density Accumulation

In this section, We will discuss our algorithm for forward density accumulation (Problem 1). To make it easy to explain, we normalize the size of a grid to 1×1. Therefore, we simplify our equation to $\rho_{x,y}^{grid} = \sum_{v \in V} w_v^{grid} \times OA(Box_v, g_{x,y}), \forall g_{x,y} \in G$.

Algorithm 1: `compute2DPrefixSum`(A)

```
1 m, n ← A.size;
2 P ← M × N zero matrix;
3 for i = 1 to M do
4 |   for j = 1 to N do
5 |   |   P_{i,j} ← P_{i,j-1} + A_{i,j};          ▷ Define P_{i,0} = 0
6 for j = 1 to N do
7 |   for i = 1 to M do
8 |   |   P_{i,j} ← P_{i-1,j} + P_{i,j};          ▷ Define P_{0,j} = 0
9 return P;
```

$$D = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad P = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Fig. 4: If we add a value 1 to $D_{2,2}$, and let P denote the 2D prefix sum of D, we will get that value propagated to the bottom-right region in P.

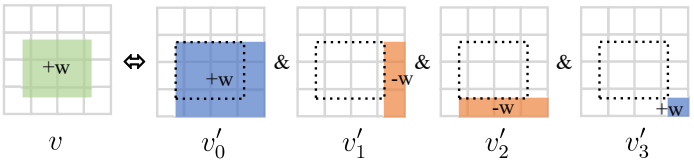


Fig. 5: An instance v is decomposed to 4 bottom-right instances v'_0, v'_1, v'_2, v'_3. The increment w on instance v is equivalent to v'_0+=w, v'_1-=w, v'_2-=w, v'_3+=w.

One straightforward method to compute forward accumulation is to find the overlapping grids for each instance and increment the grids one by one according to the overlapping areas. We denote this method as the Naïve method. One drawback of the Naïve method is that the number of increments is linear in the size of an instance. As a result, the Naïve method is extremely slow on large instances.

We show that the increments can be done more efficiently by making use of 2D prefix sum. Specifically, we can equivalently achieve the increment on a bottom-right submatrix by first incrementing a single value in a matrix and then computing the 2D prefix sum, as shown in Figure 4.

Based on this idea, we develop a two-phase algorithm consisting of density collections for instances and a 2D prefix sum. We decompose the forward density accumulation into a set of increments on bottom-right submatrices in two steps. The first step is shown in Figure 5, where we decompose an instance into 4 bottom-right instances (i.e., instances at the bottom-right corner of the grid system). It is easy to show the equivalence of the increment on the original instance and the respective increments/decrements on the 4 bottom-right instances. If the resulting bottom-right instances have integer coordinates, then it is a bottom-right submatrix. Otherwise, we further decompose the bottom-right instances through the second step shown in Figure 6, in which the bottom-right instance is split into 4 instances, each can be represented as increments on bottom-right submatrices, or on a single grid that can be handled individually.

We present our algorithm for increments on a bottom-right instance in Algorithm 2, which is essentially the implementation of Figure 6. Lines 1 and 2 computes p_x and p_y, which correspond to the respective 0.4 and 0.6 in Figure 6. Line 3 computes instance $v''_{1(1)}$. Line 4 computes instance $v''_{2(1)}$. Line 5 computes instance v''_0, $v''_{1(2)}$ and $v''_{2(2)}$ together because they cover the same submatrix. Finally, line 6 computes instance v''_3 by adding it directly to the grid in the result matrix.

Based on Algorithm 2, Algorithm 3 provides the full procedure for forward density accumulation. The definitions of matrices A and D are provided in Algorithm 2. We perform the increments on bottom-right instances at lines 6-9 for respective v'_0-v'_3 in Figure 5. After processing all instances, we obtain matrices A and D. A stores the individual increment on single grids, and D stores the increments on bottom-right submatrices prior to the 2D prefix sum. We obtain the final result by adding A with the 2D prefix sum of D.

One drawback for the proposed prefix sum method is that it makes more increments for small instances than the Naïve method does, as a small instance may cover only a few grids. Hence, we introduce an

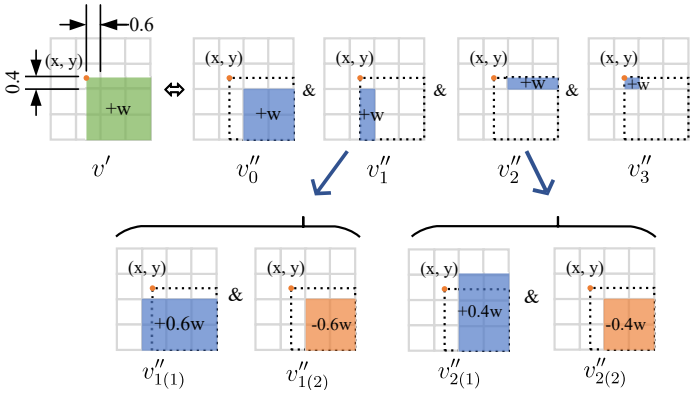


Fig. 6: An increment w on a bottom-right instance v' is split into four instances $v''_0, v''_1, v''_2, v''_3$. Of them, v''_0 is itself a bottom-right submatrix increment; v''_1 and v''_2 can be each split into two bottom-right submatrix increments; v''_3 consists of a single grid and can be handled individually.

Algorithm 2: `addToRect`(A, D, x, y, w)

Input: A, result matrix
Input: D, temporary matrix storing increments on bottom-right submatrices
Input: (x, y), the top-left coordinates of the bottom-right instance
Input: w, instance weight

1 $p_x \leftarrow \lceil x \rceil - x$;
2 $p_y \leftarrow \lceil y \rceil - y$;
3 $D_{\lceil x \rceil, \lceil y \rceil - 1}$ += $p_y w$;
4 $D_{\lceil x \rceil - 1, \lceil y \rceil}$ += $p_x w$;
5 $D_{\lceil x \rceil, \lceil y \rceil}$ += $(1 - p_x - p_y) w$;
6 $A_{\lceil x \rceil - 1, \lceil y \rceil - 1}$ += $p_x p_y w$;

Threshold to choose between the two methods based on the areas of instances.

Algorithm 3 runs in $O(MN + |V|)$, where $|V|$ is the number of instances. In other words, it is linear in both the size of the grid and the number of instances. For each instance, we break the problem into 4 calls to `addToRect`, independent to the size of the instance or how many grids covered by the instance. Therefore, our algorithm has a balanced workload for each instance.

C. Backward Density Accumulation

In this section, we propose our algorithm for backward density accumulation (Problem 2). Similar to the forward density accumulation, We also normalize the size of the grid to 1×1, and rewrite the Equation 3 as $\rho_v^{inst} \times area_v^{inst} = \sum_{g_{x,y} \in G} w_{x,y}^{grid} \times OA(Box_v, g_{x,y})$. We simplify this problem and denote the left-hand-side of this equation as $R_v^{inst} = \rho_v^{inst} \times area_v^{inst}, \forall v \in V$.

Backward density accumulation sums the weights of grids covered by each instance weighted the overlapping area ratio. We denote P as the 2D prefix sum of grid weight matrix $w^{grid} \in \mathcal{R}^{M \times N}$. Different from the procedure of the forward algorithm, here we first compute the prefix sum and then perform density collection on the prefix sum matrix. A similar idea was proposed by Crow *et al* [16] for integral images, but 1) their method only supports backward accumulation while we generalize the idea to forward accumulation, and 2) they cannot deal with non-integer coordinates, where an instance can partially cover a grid, i.e., not overlapping a full grid. The key idea of our algorithm is to decompose the summation into some combinations of the elements of 2D prefix sum matrix P and grid weight matrix w^{grid}.

Algorithm 3: `forwardAccumulation`(M, N, V)

1 $A \leftarrow M \times N$ zero matrix;
2 $D \leftarrow M \times N$ zero matrix;
3 **for** *each instance* $v \in V$ **do**
4 $(x_1, y_1, x_2, y_2) \leftarrow Box_v$;
5 **if** *Area of* $Box_v \geq$ *Threshold* **then**
  `/* PrefixSum method */`
6 `addToRect`($A, D, x_1, y_1, w_v^{inst}$);
7 `addToRect`($A, D, x_1, y_2, -w_v^{inst}$);
8 `addToRect`($A, D, x_2, y_1, -w_v^{inst}$);
9 `addToRect`($A, D, x_2, y_2, w_v^{inst}$);
10 **else**
  `/* Naïve method */`
11 **for** $i = \lfloor x_1 \rfloor$ *to* $\lfloor x_2 \rfloor$ **do**
12 **for** $j = \lfloor y_1 \rfloor$ *to* $\lfloor y_2 \rfloor$ **do**
13 Denote $g_{i,j}$ as region (i, j),$(i+1, j+1)$;
14 $A_{i,j}$+=$w_v^{inst} OA(Box_v, g_{i,j})$
15 $P \leftarrow$`compute2DPrefixSum`(D);
16 **return** A + P;

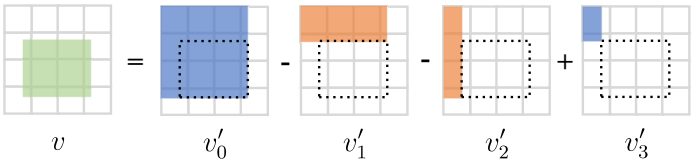


Fig. 7: A instance v can be decomposed to four top-left instances: v'_0, v'_1, v'_2, and v'_3.

As illustrated in Figure 7, we decompose an instance v into four independent *top-left instances*: v'_0, v'_1, v'_2, and v'_3, each of which is a rectangle starting at $(0, 0)$. There are two scenarios to consider for top-left instances v'. In one case, if the coordinates of the bottom-right corner (x, y) of v' happens to be integers, it is exactly the element $P_{x,y}$ in P. Otherwise, we further decompose the top-left special box sum v' into four terms: v''_0, v''_1, v''_2, and v''_3 as illustrated at Figure 8. Each term can be computed from the prefix sum matrix P or grid weight matrix w^{grid}. Similar to that in forward density accumulation, v''_0 itself is a prefix sum element in P; v''_1 and v''_2 can be further decomposed into a weighted subsection of two prefix sum elements in P; v''_3 is within one grid and can be computed from grid weight matrix w^{grid}.

The routine for computing top-left instance is summarized in Algorithm 4 corresponding to Figure 8. Line 1 and 2 computed the overlapping area ratio p_x and p_y, which correspond to 0.6 and 0.5 in Figure 8. Line 3-6 compute for term v''_0, v''_1, v''_2, and v''_3, respectively. Finally, we combine all these four terms and get the result of a top-left instance.

Based on Algorithm 4, we present our algorithm for solving the backward density accumulation in Algorithm 5. We first compute the 2D prefix sum of w^{grid} (line 1), and then compute the results for 4 top-left instances (line 5). Similarly to that in the forward density accumulation, we introduce a *Threshold* for small instances, as the Naive method has a lower overhead to compute them (lines 7-11). Algorithm 5 runs in $O(MN + |V|)$, the same as Algorithm 3 for the forward case.

D. Parallelization and GPU Acceleration

In this section, we introduce our CPU and GPU parallelization for the algorithms. we parallelize 2D prefix sum (Algorithm 1) by allocating threads for each row (lines 3-5) to finish the 1D prefix sum at rows, and each column (lines 6-8) for 1D prefix sum at columns.

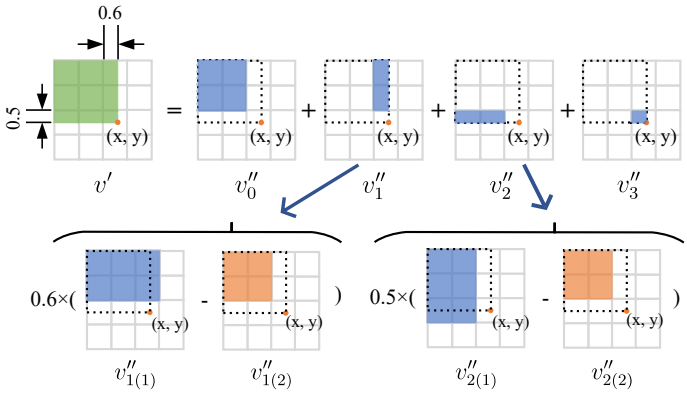


Fig. 8: A top-left instance is equivalent to the sum of 4 terms: v''_0, v''_1, v''_2, and v''_3. Given the grid weight matrix w^{grid} and 2D prefix sum matrix P, each term can be further computed in constant time.

Algorithm 4: `topLeftSum`(w^{grid}, P, x, y)

1 $p_x \leftarrow x - \lfloor x \rfloor$;
2 $p_y \leftarrow y - \lfloor y \rfloor$;
3 $v'_0 \leftarrow P_{\lfloor x \rfloor, \lfloor y \rfloor}$;
4 $v'_1 \leftarrow p_x(P_{\lfloor x \rfloor+1, \lfloor y \rfloor} - P_{\lfloor x \rfloor, \lfloor y \rfloor})$;
5 $v'_2 \leftarrow p_y(P_{\lfloor x \rfloor, \lfloor y \rfloor+1} - P_{\lfloor x \rfloor, \lfloor y \rfloor})$;
6 $v'_3 \leftarrow p_x p_y w^{grid}_{\lfloor x \rfloor+1, \lfloor y \rfloor+1}$;
7 **return** $v'_0 + v'_1 + v'_2 + v'_3$

For the density collection phase of both forward and backward density accumulation, we can parallelize the processing of instances as they are mostly independent. However, as one grid may be simultaneously updated by multiple threads in the forward accumulation, we need the `atomicAdd` primitive to avoid data race. We sketch the kernels for the forward accumulation in Algorithm 6 for GPU. The implementation for the backward accumulation is similar, so we skip it for brevity.

IV. Experimental Results

We implement our density accumulation kernels in C++ and CUDA and evaluated the performance using large designs from ISPD 2005 & 2015 contests [10], [15]. We undertake experiments on a 64-bit Linux machine with 20 cores Intel Xeon CPU at 2.10GHz, 256GB RAM and 1 GeForce RTX 2080 GPU. We adopt double-precision floating-point numbers when running the kernels, and measure the time using an average of ten runs for each setting. The number of threads in a GPU block is set to T=64, and the *Threshold* in Section III is set to 4.

We compare our density accumulation kernel with four kernels: SEQ, Naive, Gessler, and MTPerInst, some of which are used in the state-of-the-art placers. SEQ stands for the Naive method running at a single thread, which is equivalent to the sequential implementation. For forward accumulation, Gessler [14] introduces thread-local copies of resulting matrices to eliminate synchronization on CPU; MTPerInst [8] introduces instance-level parallelization on a GPU target by allocating 4 threads for each instance. For backward accumulation, we compare our implementation with Naive only, because Gessler and MTPerInst are designed specifically for forward accumulation.

We validate our kernels on two scenarios: bounding boxes of cells and nets (corresponding to small and large instances). We dump the bounding boxes and the grid settings from different global placement iterations of an open-source nonlinear placement engine [8], i.e., {0, 50, 100, 200, 300, $\cdots$, 800}, as the global placement usually finishes at

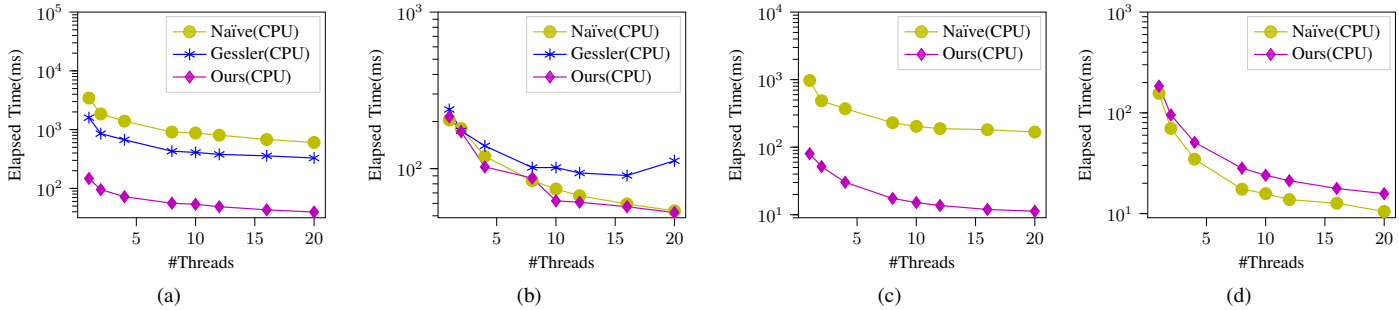


Fig. 9: The elapsed time of forward/backward density accumulation w.r.t. the number of threads on `bigblue3` with different accumulation strategies. Forward on (a) net and (b) cell instances; backward on (c) net and (d) cell instances

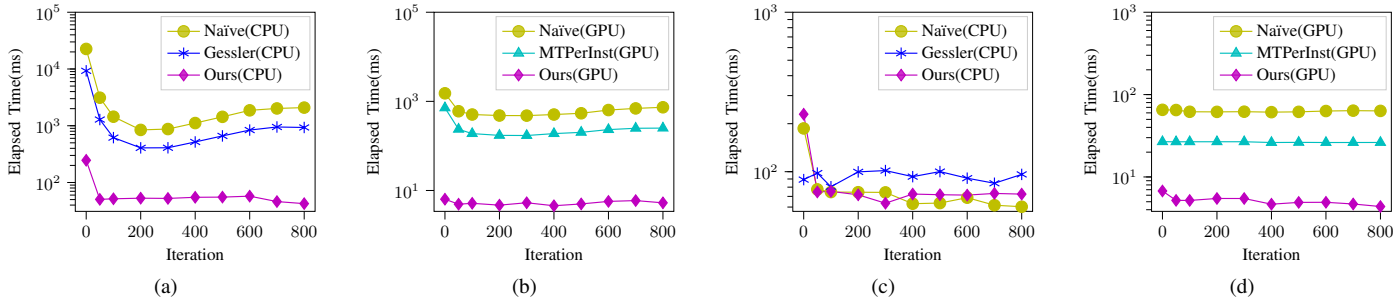


Fig. 10: The elapsed time of forward density accumulation at different placement iterations on `bigblue3` with different accumulation strategies. Net instances on (a) CPU and (b) GPU; cell instances on (c) CPU and (d) GPU.

Algorithm 5: `backwardAccumulation`(w^{grid}, V)

1 $P \leftarrow$`compute2DPrefixSum`(w^{grid});
2 **for** *each instance* $v \in V$ **do**
3 $(x_1, y_1, x_2, y_2) \leftarrow Box_v$;
4 **if** *Area of* $Box_v \geq$ *Threshold* **then**
`/* PrefixSum method */`
5 $R_v^{inst} \leftarrow$`topLeftSum`(w^{grid}, P, x_2, y_2) - `topLeftSum`(w^{grid}, P, x_1, y_2) - `topLeftSum`(w^{grid}, P, x_2, y_1) + `topLeftSum`(w^{grid}, P, x_1, y_1);
6 **else**
`/* Naïve method */`
7 $R_v^{inst} \leftarrow 0$;
8 **for** $i = \lfloor x_1 \rfloor$ *to* $\lfloor x_2 \rfloor$ **do**
9 **for** $j = \lfloor y_1 \rfloor$ *to* $\lfloor y_2 \rfloor$ **do**
10 Denote $g_{i,j}$ as region (i, j),$(i+1, j+1)$;
11 R_v^{inst} += $w_{i,j}^{grid} OA(Box_v, g_{i,j})$;
12 **return** $R_v^{inst}, \forall v \in V$;

Algorithm 6: `forwardAccumulation` on GPU

1 **function** `forwardInstKernel`(*A, D, Box,* w^{inst}, *L*) **:**
2 $j \leftarrow$ **blockIdx**.$x \times$ **blockDim**.$x +$ **threadIdx**.$x + 1$;
3 **if** $j > L$ **then return**;
4 $v \leftarrow$ the j^{th} instance in V;
5 $(x_1, y_1, x_2, y_2) \leftarrow Box_v$;
6 Compute density of v, same as Algorithm 3 lines 4-14;
7 $A, D \leftarrow M \times N$ zero matrices;
8 Call `forwardInstKernel`(*A, D,Box,* w^{inst}, $|Box|$) with $\lceil \frac{|Box|}{T} \rceil$ blocks and T threads;
`/* Call 2D prefix sum on GPU */`
9 $P \leftarrow$`compute2DPrefixSum`(D);
10 **return** A+P;

around 800 iterations. The runtime values are averaged to one iteration for each benchmark if not specially mentioned.

Table II lists the benchmark statistics for net boxes and the overall performance comparison. On average, our CPU kernel for forward density accumulation is faster than Naive and Gessler by 51.90× and 22.95×, respectively. Our GPU kernel achieves 8.62× speedup compared to our CPU kernel, and 181.19×, 64.83× compared to GPU kernels of Naive and MTPerInst, respectively. For backward density accumulation, we achieve 41.55× speedup on CPU compared to Naive, and 209.35× on GPU. We ascribe the large runtime gains on net boxes to our constant time processing for each instance, which is very suitable for computing density accumulation on large boxes; e.g., the table shows that the size of a net box is equivalent to around 1000 grids on average.

Table III lists the performance comparison on cell boxes. Our CPU kernel is 1.24× faster than Gessler, but 0.94× slower than Naive. This is because cell boxes are much smaller than net boxes, the sizes of which typically take 2-3 grids. For such kind of small instances, the Naive method requires fewer operations for each instance than our method. In addition, computing the prefix sum usually takes 20% of the runtime for cell instances, which is an additional overhead. However, our algorithm is highly parallelizable on GPU because of its constant balanced workload for instances, while the Naive method has an imbalanced workload for different box sizes. For example, we achieve 15.91× speedup by migrating our kernels to GPU, making our kernel 4.49× and 2.15× faster than the GPU implementations of Naive and MTPerInst. We observe similar patterns in backward case, where we are 0.74× slower than Naive on CPU but 4.75× faster on GPU compared to the Naive GPU kernel. Generally, our algorithm is faster for larger cells, making it more suitable for accelerating macro placement

TABLE II: Runtime comparison on nets. SEQ runs with 1 thread; other CPU kernels run with 20 threads.

Benchmark	Statistics			CPU Forward				GPU Forward			CPU Backward			GPU Backward	
	#Nets	Box Size†		SEQ	Naïve	Gessler	Ours	Naïve	MTPerInst	Ours	SEQ	Naïve	Ours	Naïve	Ours
		Avg	Max												
bigblue1	284K	1.78K	1048K	2018.75	626.62	295.21	11.59	440.57	92.46	1.73	527.74	142.69	2.25	433.89	1.72
bigblue2	577K	2.23K	1027K	2533.84	466.57	232.12	17.53	204.10	96.85	2.22	661.85	111.99	3.37	329.87	2.05
bigblue3	1123K	5.10K	4018K	9875.79	3559.61	1528.88	56.76	674.42	266.86	4.99	2103.98	435.18	10.39	1005.33	5.64
bigblue4	2229K	6.95K	4158K	26362.47	4499.48	1895.84	93.01	2076.18	731.44	7.81	6654.99	771.46	14.57	2113.05	7.87
superblue12	1293K	0.75K	1495K	3972.19	964.49	451.10	24.29	539.80	199.46	4.22	1400.83	153.78	6.55	693.03	4.16
superblue14	619K	2.01K	580K	3938.26	1258.69	584.47	10.86	463.97	168.80	2.34	1056.26	130.15	3.31	470.23	2.16
superblue16	697K	2.43K	316K	4370.09	717.17	321.55	15.88	463.68	187.66	2.66	1294.14	154.46	3.95	630.35	2.37
superblue19	511K	0.78K	300K	843.77	211.32	132.84	7.21	120.51	39.44	1.53	245.39	33.83	2.12	121.97	1.72
Avg. Ratio				1960.40	447.38	197.88	8.62	181.19	64.83	1.00	503.54	69.82	1.68	209.35	1.00

TABLE III: Runtime comparison on cells. SEQ runs with 1 thread; other CPU kernels run with 20 threads.

Benchmark	Statistics			CPU Forward				GPU Forward			CPU Backward			GPU Backward	
	#Cells	Box Size†		SEQ	Naïve	Gessler	Ours	Naïve	MTPerInst	Ours	SEQ	Naïve	Ours	Naïve	Ours
		Avg	Max												
bigblue1	636K	2.40	10K	45.91	15.78	22.43	17.55	3.85	2.13	1.73	26.70	1.78	2.80	3.74	2.12
bigblue2	1496K	2.27	5K	104.37	32.05	32.22	29.95	2.67	2.82	2.61	64.78	4.17	5.35	1.80	2.89
bigblue3	2042K	3.42	329K	198.63	78.75	108.52	78.13	63.10	26.66	5.54	122.13	10.17	15.50	59.01	5.25
bigblue4	5062K	2.33	125K	456.73	98.66	144.74	108.37	32.45	13.20	10.13	317.28	20.75	27.75	28.32	7.49
superblue12	1463K	2.03	20K	81.57	87.74	57.50	93.21	5.31	4.29	2.74	49.41	3.41	3.80	5.02	1.85
superblue14	682K	2.12	3K	39.60	27.16	54.13	28.69	2.85	1.84	1.29	23.76	1.88	2.13	2.78	1.34
superblue16	734K	2.13	3K	42.34	41.08	54.82	44.59	2.61	2.38	1.41	25.67	2.04	2.14	2.48	1.35
superblue19	601K	2.21	42K	35.24	21.80	53.40	23.73	7.00	3.93	1.21	21.06	1.60	1.85	8.45	1.21
Avg. Ratio				37.67	15.11	19.79	15.91	4.49	2.15	1.00	27.70	1.95	2.61	4.75	1.00

†Box size is defined as the number of grids a box covers, which is computed by dividing the box area by the area of a grid.

with movable large macros and standard cells.

Figure 9 demonstrates the scalability of different CPU kernels. We observe that the algorithms gradually saturate at 16-20 threads. Gessler on cell boxes even experiences slow down on 20 threads, as shown in Figure 9(b). One possible reason is the heavy memory overhead for keeping thread-local copies of the resulting matrices. Our method maintains high performance on both forward/backward accumulation of net boxes, and forward accumulation on cell boxes. However, we are slightly slower than Naive on cell boxes in the backward case, as shown in Figure 9(d), due to the small size of cell instances.

Figure 10 draws the runtime for forward density accumulation w.r.t the iterations of a placement process. Higher runtimes are observed at the beginning of the iterations because of the highly overlapping initial cell placement. With cells gradually spreading out, the runtime drops quickly. Gessler shows good performance on cell boxes at early iterations, as it does not require data synchronization, but the benefits no longer exist at later iterations and the memory overhead becomes dominating when cells spread out, as shown in Figure 10(c).

V. Conclusion

In this paper, we have proposed a set of new kernels for accelerating density accumulation on CPU/GPU. We decompose the problem into constant-time density collection for each instance and a linear-time 2D prefix sum, and develop CPU and GPU parallelization for our algorithms. Compared to kernels used in state-of-the-art placers, we achieved up to 22× speed-up on CPU and 64× on GPU. Our future work includes further reducing runtime overhead by introducing instance-level parallelization, as well as exploring Single Instruction Multiple Data (SIMD) instructions in 2D prefix sum.

Acknowledge

This work was supported in part by the National Science Foundation of China (Grant No. 62004006 and No. 62034007), Beijing Municipal Science and Technology Program (Grant No. Z201100004220007), and Zhejiang Provincial Key R&D program (Grant No. 2020C01052).

References

[1] T. Lin, C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "POLAR: A high performance mixed-size wirelengh-driven placer with density constraints," *IEEE TCAD*, vol. 34, no. 3, pp. 447–459, 2015.

[2] 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.

[3] 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.

[4] J. Lu, P. Chen, C.-C. Chang, L. Sha, D. J.-H. Huang, C.-C. Teng, and C.-K. Cheng, "ePlace: Electrostatics-based placement using fast fourier transform and nesterov's method," *ACM TODAES*, vol. 20, no. 2, p. 17, 2015.

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

[6] A. B. Kahng and Q. Wang, "A faster implementation of APlace," in *Proc. ISPD*. ACM, 2006, pp. 218–220.

[7] T. F. Chan, K. Sze, J. R. Shinnerl, and M. Xie, "Mpl6: Enhanced multilevel mixed-size placement with congestion control," in *Modern Circuit Placement*. Springer, 2007.

[8] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," *IEEE TCAD*, June 2020.

[9] C.-X. Lin and M. D. Wong, "Accelerate analytical placement with gpu: A generic approach," in *Proc. DATE*. IEEE, 2018, pp. 1345–1350.

[10] 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.

[11] C.-L. E. Cheng, "RISA: Accurate and efficient placement routability modeling," in *Proc. ICCAD*, 1994, pp. 690–695.

[12] P. Spindler and F. M. Johannes, "Fast and accurate routing demand estimation for efficient routability-driven placement," in *Proc. DATE*, 2007, pp. 1226–1231.

[13] N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei, "The dac 2012 routability-driven placement contest and benchmark suite," in *Proc. DAC*. IEEE, 2012, pp. 774–782.

[14] F. Gessler, P. Brisk, and M. Stojilovič, "A shared-memory parallel implementation of the replace global cell placer," in *Proc. VLSI Design*. IEEE, 2020, pp. 78–83.

[15] 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.

[16] F. C. Crow, "Summed-area tables for texture mapping," in *SIGGRAPH '84*. New York, NY, USA: ACM, 1984, p. 207–212.