A Shared-Memory Parallel Implementation of the RePlAce Global Cell Placer

Frédéric Gessler*, Philip Brisk† and Mirjana Stojilović*
*École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
†University of California Riverside (UCR), Riverside, USA
Email: frederic.gessler@epfl.ch, mirjana.stojilovic@epfl.ch

***Abstract*—RePlAce is a state-of-the-art prototype of a flat, analytic, and nonlinear global cell placement algorithm, which models a placement instance as an electrostatic system with positively charged objects. It can handle large-scale standard-cell and mixed-cell placement, while achieving shorter wirelength and similar or shorter runtimes than other state-of-the-art placers on the ISPD-2005/2006 standard-cell benchmarks; however, the runtime of RePlAce on these benchmarks ranges from 15 minutes to 5+ hours on a 2.6 GHz Intel Xeon server running a single thread, rendering development cycles prohibitively long. To address this concern, this paper introduces a multi-threaded shared-memory implementation of RePlAce. The contributions include techniques to reduce memory contention and to effectively balance the workload among threads, targeting the most substantial performance bottlenecks. With 2–12 threads, our parallel RePlAce speeds up the bin density function by a factor of 4.2–10×, the wirelength function by a factor of 2.3–3×, and the cost gradient function by a factor of 2.9–6.6× compared to the single-threaded original RePlAce baseline. Moreover, our parallel RePlAce is ≈3.5× faster than the state-of-the-art PyTorch-based placer DREAMPlace, when both are running on 12 CPU cores.**

***Index Terms*—VLSI placement, multithreading, parallelism**

I. Introduction

Placement is an important step in VLSI design, directly impacting timing closure, die utilization, routability, and design turnaround time [1]. Among all academic placers, recent electrostatic-based global placers, such as ePlace and RePlAce, achieve the best known results in terms of half-perimeter wirelength [2], [3]. ePlace uses electrostatics-based global-smooth density cost function, which is solved numerically by a fast Fourier transform (FFT). The nonlinear optimization is solved using Nesterov's method [4], which provides faster convergence, using the Lipschitz constant to dynamically predict the step length. RePlAce extends ePlace with a fast nonlinear engine for standard-cell, mixed-size, and 3D-IC mixed-size placement, achieving superior results and a lower runtime than ePlace; however, RePlAce still requires hours to legally place industrial-grade circuits. Recent placer DREAMPlace considerably reduces the runtime using PyTorch's engine, but, for the moment, it runs efficiently only on GPUs [5].

To address the runtime concern on generic and most widely available CPU-based hardware platforms (multicore CPUs), we propose and design an efficient parallel implementation of RePlAce that uses the OpenMP shared memory parallel programming library [6]. The contributions of the paper are summarized as follows:

- The first detailed exploration of strategies to accelerate RePlAce via shared memory parallelism.
- The acceleration of global placement by reducing memory traffic through cache-aware data structures, privatization of the overlap sum in the bin density, loop fission in the cost gradient function, and workload balancing in the wirelength and wirelength gradient functions.
- At 12 CPU threads, ≈3× speedup over RePlAce [3] and ≈3.5× speedup over DREAMPlace [5] running on CPU.

The rest of the paper is organized as follows. Section II summarizes related work. Section III-A formally introduces the global placement (GP) in RePlAce. Section IV identifies the phases in GP that dominate the execution time and describes our approach to reduce memory traffic. Section V details our parallel RePlAce implementation, while Section VI presents experimental results. Section VII concludes the paper.

II. Related Work

VLSI placers use analytical models, optimizing a global convex wirelength function with multiple objectives, or quadratic models. Due to high runtime, their acceleration is an active research topic.

Even though RePlAce is an analytic placer [3], shared-memory parallel implementations of quadratic placers are relevant to our work. Parallel SimPL [7] optimizes throughput by using compressed sparse row matrix multiplication and by exploiting Intel x86 SSE vector instructions. It achieves speedup of up to 2.4× with 8 threads for global placement on the ISPD-2005 benchmarks [7], but fails to scale effectively due to memory contention. POLAR 3.0, on the other hand, is based on a divide-and-conquer cell partitioning scheme [8]. It reports speedup of up to 4.2× with 16 threads for global placement on ISPD-2005 benchmarks.

Recent efforts to accelerate analytical placers focus on using GPUs. For instance, Lin et al. [9] represent the circuit as a sparse graph and employ five GPU kernels that convert the wirelength gradient computation into a sequence of sparse matrix multiplications and vector operations, achieving a speedup of about 170×. In RePlAce, wirelength gradient accounts for about one third of the runtime; hence, this speedup would enable ≈1.5× faster placement. Their GPU-accelerated bin density yields speedups from 4.3× to 7×. The most recent GPU-placer is DREAMPlace [5]. It is built on the idea of using features from machine learning frameworks (PyTorch

specifically), such as the nonlinear optimization engine. It achieves speedup of up to 30× on global placement by casting analytic placement into neural net training and implementing the relevant custom CUDA operators. While its architecture would allow to perform the forward and backward propagation in shared memory, a multithreaded implementation of the operators is currently lacking and the performance of DREAMPlace is not yet satisfactory on CPU nodes. While GPUs have proven to be a fit architecture for accelerating analytical placement, the off-the-shelf hardware needed to obtain such speedups is typically more expensive than a standard ≈8-cores CPU.

Other prior work includes GPU-based implementations of TimberWolf [10] and quadratic placers [11], [12], and shared-memory implementations of placement legalization [13].

III. Background

A. Analytical Placement

A placement instance is a hyper-graph $G = (V, E, R)$, where V is a set of vertices (**cells** and macros), E a set of hyper edges (**nets**), and R the placement region decomposed into $m \times m$ rectangular grids (**bins**). A placement solution is a pair of integer sets $\mathbf{v} = \{x_1, \ldots, x_n, y_1, \ldots, y_n\}$, where (x_i, y_i) is the placed location of the origin of the cell c_i and n is the total number of cells. A legal placement solution assigns vertices to bins, horizontally aligned with the boundaries of a placement row and without overlapping other cells.

Placement is typically performed in three steps: initial, global, and detailed placement. The initial placement aims to minimise the wirelength unconstrained by density and legality concerns. The detailed placement legalises the design after the global placement. RePlAce, the focus of this paper, is a global placer and the most computationally intensive of the three steps. Both RePlAce and our work use an established quadratic placer [14] in the initial placement and a well-known detailed placer [15], without modification.

B. Global Placement with RePlAce

The objective of global placement is to minimize the **half-perimeter wirelength** (HPWL) of all nets $e \in E$:

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

Here, i and j are net pins, while x and y are pin coordinates. Since HPWL is not differentiable, numerous smoothing strategies have been proposed in the past [2]. RePlAce uses weighted-average strategy [16], which approximates the wirelength as

$$\tilde{W}_x(e) = \frac{\sum_{i \in e} x_i \exp(x_i/\gamma)}{\sum_{i \in e} \exp(x_i/\gamma)} - \frac{\sum_{i \in e} x_i \exp(-x_i/\gamma)}{\sum_{i \in e} \exp(-x_i/\gamma)}, \quad (2)$$

where parameter γ controls the smoothness and accuracy of the approximation. The total wirelength $\tilde{W}(\mathbf{v})$ (the sum of $\tilde{W}_x(\mathbf{v})$ and $\tilde{W}_y(\mathbf{v})$) is smooth and differentiable.

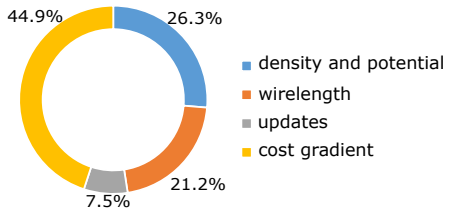

Fig. 1. GP breakdown, averaged over ISPD-2005/2006 benchmark circuits.

Given a placement instance $\mathbf{v}$, its **bin density** $\rho_b(\mathbf{v})$ equals

$$\rho_b(\mathbf{v}) = \sum_{i \in V} l_x(b,i) l_y(b,i), \quad (3)$$

where $l_x(b,i)$ and $l_y(b,i)$ are the horizontal and vertical area overlaps between bin b and cell c_i. Besides minimizing HPWL, global placement must achieve a density below a user-specified target density. As a consequence, the global placement objective can be defined as the cost function

$$\min_{\mathbf{v}} f(\mathbf{v}) = \tilde{W}(\mathbf{v}) + \lambda N(\mathbf{v}). \quad (4)$$

Here, N is the density penalty and the penalty factor λ balances the influence of wirelength and density on the cost function. Unlike prior analytical placers [17], RePlAce models the density penalty using an electrostatic analogy [2], in which a cell c_i is modeled as a positively charged particle whose electric charge equals the cell area. To obtain the system potential energy (density) and the electric field (density gradient), RePlAce uses an FFT library that numerically solves a Poisson's equation [2], [3]. Finally, RePlAce adopts Nesterov's method to solve the cost optimization problem.

IV. RePlAce Algorithm and Analysis

We present here a detailed runtime analysis of RePlAce. Then, we introduce new data structures to improve memory access efficiency. Our parallel implementation of RePlAce, described in the next section, employs these new data structures.

A. Runtime Analysis

We ran RePlAce on the ISPD-2005/2006 benchmarks and aggregated the runtime across all of them. According to the results, global placement is the main bottleneck, accounting for more than 80% (≈81.7%) of total runtime. Fig. 1 shows that the most time-consuming global placement tasks are computing the gradient of the cost function (≈45%), computing the weighted average of the wirelength (≈21.2%), computing the bin density and electric potential (≈26.3%). Updating the placement, i.e., moving the cells and updating the step length, takes the least amount of time (≈7.5%) and we will thus ignore it. Bin density computation takes slightly longer (≈55%) than solving the potentials and electric fields using an FFT.

Our work parallelizes $44.9\% + 0.55 \cdot 26.3\% + 21.2\% = 80.5\%$ of the entire global placement flow or, alternatively, $80.5\% \cdot 81.7\% = 65.7\%$ of the total CPU time. According

TABLE I
OUR SLIM DATA TYPES, ALLOWING REDUCED MEMORY TRAFFIC. ALL STRUCTURES FIT INTO A STANDARD 64-BYTE WIDE CACHE LINE.

fpos2_t	**pin_t**	**net_t**	**cell_phy_t**	**cell_den_t**	**area_t**	**bin_t**
`float` *x*	`fpos2_t` *expL*	`pin_t` *pinArray[]*	`pin_t**` *pinArrayPtr*	`pos2_t` *binStart*	`pos2_t` *coord*	`fpos2_t` *min*
`float` y	`fpos2_t` *expR*	`fpos2_t` *sumNumL*	`int` *pinCNT*	`pos2_t` *binEnd*	`long int` *terminArea*	`fpos2_t` *max*
Size: 8 bytes	`fpos2_t` *coord*	`fpos2_t` *sumNumR*	`char` *type*	`fpos2_t` *min*	`float` *virtArea*	`fpos2_t` *field*
	`int` *pinID*	`fpos2_t` *sumDenL*	**Size: 16 bytes**	`fpos2_t` *max*	`float` *binDensity*	`float` *cellArea*
pos2_t	`int` *moduleID*	`fpos2_t` *sumDenR*		`fpos2_t` *size*	`float` *fillerDensity*	`float` *fillerArea*
`int` *x*	`int` *netID*	`fpos2_t` *min*		`float` *scale*	**Size: 32 bytes**	`float` *potential*
`int` *y*	`char` *metaData1*	`fpos2_t` *max*		`char` *type*		**Size: 36 bytes**
Size: 8 bytes	`char` *metaData2*	`int` *pinCNT*		**Size: 48 bytes**		
	Size: 40 bytes	**Size: 64 bytes**				

to Amdahl's law, the maximum achievable speedup is 2.91×. For T threads, the ideal speedup is

$$S(T) = \left(34.3\% + \frac{65.7\%}{T}\right)^{-1}. \quad (5)$$

Consequently, the maximum speedup for 2–12 threads is 1.49–2.51×. We, however, achieve even higher speedups, because we first address high memory traffic and poor data locality.

B. Reducing Memory Traffic

Detailed profiling of the wirelength, bin density, and cost gradient computations revealed that all three are memory-bound. Hence, our first acceleration strategy, prior to parallelization, is to reduce the memory traffic and increase data locality. Our solution is to introduce new data structures (see Table I) into RePlAce, which are subsets of the original data structures and which can fit into a standard 64-byte cache line. They are carefully designed to optimize for data locality in the placement functions that access them frequently.

1) Pins in Placement Instance: Our first strategy towards increasing data locality is to minimize the memory traffic incurred while computing the wirelength $\tilde{W}(\mathbf{v})$, by defining a new data structure **pin_t**, which holds the pin data close to the wirelength-related data that would otherwise have to be frequently recomputed. Therefore, **pin_t** fields are pin IDs, the cells to which the pins belong ($moduleID$), and the nets whose terminal the pins are ($netID$). In addition, **pin_t** holds pin coordinates $coord$ and the exponential denominators $expL$ and $expR$, required to efficiently compute $\tilde{W}(\mathbf{v})$. Here, L and R stand for left and right, respectively, while $coord$ are instances of structure `fpos2_t` (floating point 2-D position).

2) Nets in Placement Instance: Nets are defined by the number of pins $pinCNT$ and the list of pin IDs $pinArray$. Knowing that wirelength computation requires traversing all nets and computing per-net wirelength, we keep all data that is frequently accessed—the coordinates of the net's furthest pins (min and max), and the per-net sums in numerators and denominators of (2)—as part of the same data structure **net_t**. This has another advantage: it allows computing the final wirelength as a parallel sum of partial wirelengths.

3) Cells in Placement Instance: Another new slim data type is **cell_phy_t**, containing only data required by the wirelength gradient computation: the list of cell pins, the number of cell pins, and the cell type (standard, macro, filler).

While computing bin density, cell size and location (determined by the coordinates of two extreme corners of the cell rectangle, min and max) are frequently accessed values. Hence, we choose to create a dedicated data structure **cell_den_t**, to keep close the data required for computing bin density: the location, the size ($size$), the type ($type$) and scale factor[1] of the cell. Additionally, **cell_den_t** caches the indices of two bins, *binStart* and *binEnd*, allowing the density computation function to traverse only those bins in the placement region that overlap with the given cell.

4) Bins in Placement Instance: The placement region is decomposed into an $m \times m$ grid where $m = \lceil \log_2 \sqrt{n} \rceil$ for n cells. This decomposition is tailored so that, on average, one cell occupies one bin and the total number of bins is a power of 2, which facilitates efficient FFT transpositions.

In RePlAce, a bin in the placement region is characterized by its position ($coord$), electric field ($field$), electric potential ($potential$), and area overlap with different types of objects (standard cells, fillers, terminals, virtual nodes etc.). Some of this information is loop-invariant or accessed by particular functions only. To address that, we introduce two new data structures: **area_t** and **bin_t**. The former keeps the loop-invariant information consumed by the FFT solver, while the latter keeps bin location data near to its electrical properties.

C. Original Multithreaded RePlAce

A straightforward OpenMP multithreaded implementation of RePlAce is provided by its authors [3]. It accelerates global placement by ≈2× on average, but suffers heavily from congestion. For the wirelength, it allocates one thread per each net. For the cost gradient, it forks one thread per cell, computes the preconditioned cost for that cell, and then joins. In the sections that follow, we describe our new approach to parallelizing RePlAce.

V. PARALLEL IMPLEMENTATION

A. Wirelength

Algorithm 1 shows the most computationally intensive subroutine of the wirelength computation (after rewriting the original RePlAce code to use our new slim data structures **net_t** and **pin_t**); it is the computation of the left and right exponential terms in (2), which enumerates each net in the

[1]RePlAce periodically updates the cell size by a scale factor [3]; the details and rationale for doing so are beyond the scope of this paper.

Algorithm 1 Our parallel wirelength computation. **Inputs:** Nets $\mathbf{E}$, pins $\mathbf{P}$, T threads, workload distribution $w[]$.

```
1:  #pragma omp parallel num_threads(T) {
2:    t = omp_get_thread_num()
3:    start = (t ≥ 1) ? w[t − 1] : 0
4:    end = (t < T − 1) ? w[t] : netCNT
5:    for all nets e ∈ E, start ≤ i < end do
6:      numL, numR, denL, denR = 0
7:      for all pins p ∈ e do
8:        Compute and update p.expL
9:        numL += p.coord · p.expL
10:       denL += p.expL
11:       Compute and update p.expR
12:       numR += p.coord · p.expR
13:       denR += p.expR
14:     end for
15:     e.sumNumL = numL, e.sumNumR = numR
16:     e.sumDenL = denL, e.sumDenR = denR
17:   end for
18: }
```

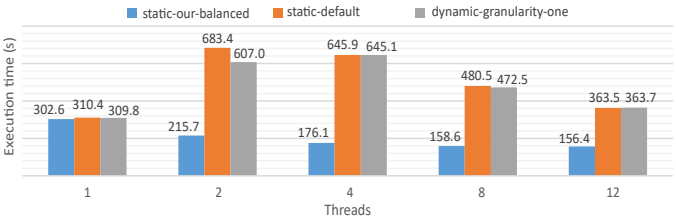

Fig. 2. The execution times of the wirelength function on the BIGBLUE4 benchmark, for our static workload balancing, default static scheduling, and dynamic scheduling with granularity equal to 1 (one net per thread).

placement instance along with all of the pins in each net. To parallelize this computation, we employ OpenMP *parallel-for* loop constructs and start with two extreme scheduling strategies: a default static scheduler that assigns an approximately equal number of nets to each thread, and a dynamic scheduler that assigns one net per thread at a time. Detailed profiling shows that the the static scheduler suffers from workload imbalance because nets have variable number of pins, while the dynamic scheduler, which exhibits better workload balance, suffers from synchronization overhead. To address this concern, we reimplemented the static scheduler to estimate the per-thread workload by the total number of pins across all nets, $\frac{1}{T}\sum_{e\in E} e.pinCNT$, as opposed to the total number of nets; here, T is the number of threads. Fig. 2 compares the performance of these strategies and shows that the updated static scheduler yields the shortest execution times. Fig. 3 provides a detailed breakdown of the per-thread workload: in most cases, it is less than 0.6% away from the mean.

B. Bin Density

Bin density is computed as the sum of the overlap surface areas between the bins and the placed cells. This necessitates a doubly-nested loop that traverses each cell and its overlapping

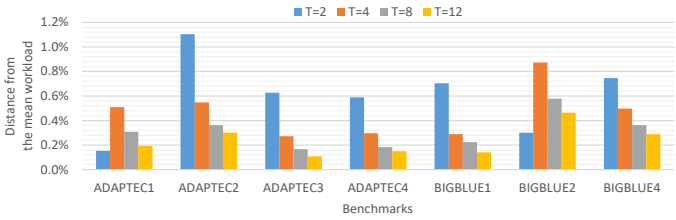

Fig. 3. The workload balance overview for ISPD-2005 benchmarks. On average, workloads are within 0.4% from the mean.

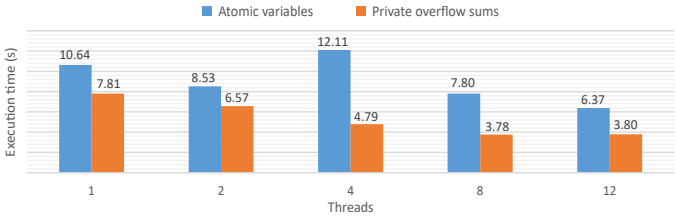

Fig. 4. The execution times of bin density computation on the ADAPTEC1 benchmark, when atomic variables or private overflow sums are used to avoid data race. Clearly, the latter yields better performance and scaling.

bins. This subroutine is memory bound, as cell-bin overlap is computed efficiently. One naive acceleration strategy is to allocate a subset of cells to each thread; however, doing so induces a data race when adding overlapping area to each bin's running sum, as two cells being processed concurrently may overlap the same bin. On the other hand, iterating over bins first would eliminate the data race, but increases synchronization overhead because the number of cells is far larger than the number of bins.

Initially, we tried to eliminate the data race using synchronization primitives: we wrapped the area sum in std-C++11 atomic variables and computed the sum using a compare-and-swap instruction[2]. Unfortunately, profiling revealed that synchronization scales poorly for more than four threads due to an increased likelihood of collisions. Instead, we privatized the computation of the overlap sum, shown in Algorithm 2. We allocate local copies of the overflow sum matrices to each worker thread. When the threads finish computation, we fork and aggregate the private results in parallel, thus eliminating data races. Fig. 4 quantifies the benefit of privatization.

To find the best scheduling strategy, we compared two approaches: naive static OpenMP scheduling and our static workload assignment. To estimate the workload per thread, we computed the average number of overlaps between cells and bins after the initial placement. However, detailed profiling revealed that, since the cells move as the algorithm advances, the workload estimated this way does not remain well balanced for long. Hence, we opted for naive static scheduling.

C. Cost Gradient

After wirelength and density computation, RePlAce runs an FFT solver to find the bin potential and electric field; the only remaining step is cost gradient computation. This task can be

[2]Ideally, we would have used a fetch_add but std-c++11 does not support floating-point fetch_add; it has been proposed for std-C++20.

Algorithm 2 Our parallel bin density computation. **Inputs:** Cells $\mathbf{V}$, bins $\mathbf{B}$, T threads, the origin of the placement instance (ORIGIN) and bins per unit of length (STEP).

```
1:  #pragma omp parallel num_threads(T) {
2:    t = omp_get_thread_num()
3:    Allocate and clear two per-thread matrices of partial
      bin-overlap sums: binOverFiller, binOverCell = 0
4:    for all cells c ∈ V do
5:      c.binStart = (c.min − ORIGIN)/STEP
6:      c.binEnd = (c.max − ORIGIN)/STEP
7:      for all bins b_i ∈ B, c.binStart ≤ i ≤ c.binEnd do
8:        minCorner = max(b_i.min, c.min),
9:        maxCorner = min(b_i.max, c.max)
10:       overlap = RectArea(minCorner, maxCorner)
11:       if c ∈ FillerCells, FillerCells ⊂ V then
12:         binOverFiller[t][i] += overlap · c.scale
13:       else
14:         binOverCell[t][i] += overlap · c.scale
15:       end if
16:     end for
17:   end for
18: }
19: #pragma omp parallel num_threads(T) {
20:   for all bins b_i ∈ B do
21:     for all t, 1 ≤ t ≤ T do
22:       b_i.fillerArea += binOverFiller[t][i]
23:       b_i.cellArea += binOverCell[t][i]
24:     end for
25:   end for
26: }
```

decomposed into three subroutines, carried out independently for all cells. First, the wirelength gradient is obtained from the precomputed exponential terms and pin coordinates cached in **`pin_t`**. Then, the density penalty is obtained similarly as in the bin density computation, but weighing the overlap area by the bin electric field. Lastly, the cost is preconditioned; this step contributes to at most 5% of total runtime, which is why we chose to leave it as-is.

The original RePlAce sequential implementation called all three subroutines in the same loop, which iterated over all cells. This call sequence could not be accelerated efficiently due to poor data locality and high memory bandwidth requirements. Our implementation uses the more efficient data structures, discussed earlier, and performs three independent loops over all cells. We parallelized the wirelength- and potential-gradient loops (Fig. 5). Similar to the wirelength function, we observed performance improvement in parallel wirelength gradient when the workload is statically balanced according to the number of cell pins. However, when computing the potential gradient, naive static scheduling worked best (Fig. 6).

VI. Experimental Results

We implemented our parallel RePlAce in C++ and built it with g++ version 7.3.0 and `-O3` optimization. We benchmark

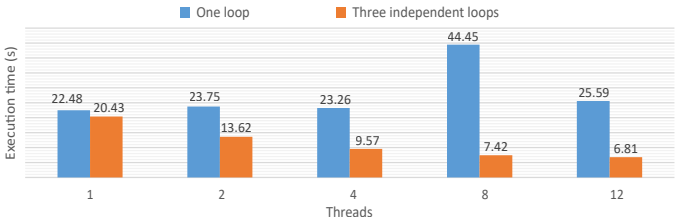

Fig. 5. The comparison of execution times of cost gradient computation on the ADAPTEC1 benchmark, showing that three independent loops scale considerably better than the original implementation.

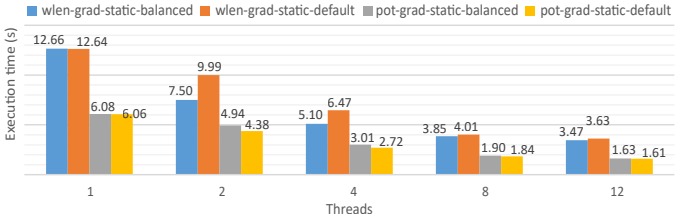

Fig. 6. The execution times of wirelength gradient and potential gradient computation on the ADAPTEC1 benchmark, when our workload-balanced static scheduling and naive static scheduling are used.

placement without dynamic step size adaptation or local density, to enable a direct comparison to DREAMPlace [5]. The initial placement uses the Eigen library with multithreading enabled; detailed placement is performed using the NTUPlace3 binary. At the time of writing, RePlAce can only place circuits without movable macros. Hence, we use all ISPD-2005 benchmarks [7] except BIGBLUE3. Our experiments are ran on an AMD Ryzen Threadripper 1920X 12-Core CPU with 64GB of DRAM@2134MHz and 32MB of LL cache.

Table II reports results for our sequential implementation of RePlACe using our proposed efficient data structures and our multithreaded parallel implementation with 1, 2, 4, 8, and 12 threads. We compare directly to RePlAce, a multithreaded version of RePlAce (12 threads) [3], and DREAMPlace (12 threads, no GPU) [5]. As an example, we reduced the time to place BIGBLUE4 benchmark by a factor of 2.36×, which amounts to 33 minutes saved. The last row indicates the difference in HPWL averaged across all benchmarks. Our results are not numerically identical due to code refactoring and reorganization, which altered the order of certain floating-point operations, along with our choice to use single-precision floating-point values in our efficient data structures; that said, the difference in observed HPWL never exceeds 0.1%.

Table III shows the speedups obtained with the same number of threads for the three subroutines that we accelerated, along with the overall speedup obtained for global placement. The introduction of slimmer data structures yields speedups of at least 1.5× for every affected function with no multithreading enabled. Two particularly successful techniques are privatization for the bin density, allowing for speedups as high as 11.99× and loop fission of the gradient function allowing for overall cost speedups as high as 7.17×.

While our proposed data structures improve memory access overhead, they do not fully eliminate the memory wall; as a

TABLE II

RUNTIME (IN SECONDS) OF THE GLOBAL PLACEMENT (GP) PHASE AND THE TOTAL TIME TO PLACE ISPD-2005 BENCHMARKS. RESULTS ARE REPORTED FOR REPLACE, REPLACE MODIFIED TO USE OUR SLIM DATA TYPES, OUR PARALLEL IMPLEMENTATIONS USING 1, 2, 4, 8, AND 12 THREADS, THE MULTITHREADED ORIGINAL VERSION OF REPLACE (12 THREADS), AND DREAMPLACE RUNNING ON A CPU WITH 12 THREADS. THE LAST TWO ROWS REPORT THE GEOMETRIC MEAN OF THE SPEEDUP COMPARED TO THE ORIGINAL SEQUENTIAL REPLACE AND THE MEAN RELATIVE ERROR OF HPWL.

	RePlAce		**Sequential**		**T=1**		**T=2**		**T=4**		**T=8**		**T=12**		**RePlAce (T=12)**		**DREAMPlace (T=12)**	
	GP	**TOTAL**	**GP**	**TOTAL**	**GP**	**TOTAL**	**GP**	**TOTAL**	**GP**	**TOTAL**	**GP**	**TOTAL**	**GP**	**TOTAL**	**GP**	**TOTAL**	**GP**	**TOTAL**
ADAPTEC1	108.68	189.30	59.60	140.21	58.68	137.97	47.57	117.91	39.43	103.64	35.59	94.64	35.22	93.33	46.45	104.07	171.13	198.63
ADAPTEC2	210.07	309.29	134.40	234.44	134.16	232.65	113.82	202.51	93.54	172.15	83.54	157.52	82.77	154.99	107.52	179.21	275.01	308.98
ADAPTEC3	443.67	628.37	294.82	481.08	298.33	484.75	231.02	395.15	179.49	328.32	151.36	290.37	144.50	282.42	215.93	352.01	457.26	520.85
ADAPTEC4	497.98	691.84	346.26	537.55	350.52	539.44	268.59	441.89	202.76	358.27	165.36	312.01	156.04	299.48	270.93	413.08	515.88	588.46
BIGBLUE1	205.09	308.17	106.40	209.47	107.72	209.15	84.17	174.03	69.56	150.98	62.83	137.58	60.88	133.85	89.25	161.91	238.53	275.14
BIGBLUE2	376.93	607.11	257.62	490.28	258.59	487.34	202.43	409.36	154.68	347.26	132.42	314.88	125.73	306.18	199.54	378.20	438.05	535.23
BIGBLUE3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1178.98	1331.79
BIGBLUE4	2461.81	3478.51	1571.31	2580.64	1675.59	2678.92	1335.57	2212.92	1003.35	1781.11	821.90	1548.15	763.90	1472.50	1073.69	1777.88	2264.48	2610.96
geomean	**1.00**	**1.00**	**1.60**	**1.33**	**1.58**	**1.33**	**1.99**	**1.59**	**2.53**	**1.88**	**2.94**	**2.10**	**3.06**	**2.17**	**2.09**	**1.78**	**0.87**	**1.13**
ΔHPWL(%)	**0.00%**		**-0.08%**		**-0.09%**		**-0.10%**		**-0.10%**		**-0.10%**		**-0.09%**		**-0.32%**		**-0.15%**	

TABLE III

INDIVIDUAL SPEEDUPS OF GP, BIN DENSITY, WIRELENGTH, AND COST GRADIENT FUNCTION, VERSUS THEIR ORIGINAL REPLACE IMPLEMENTATIONS.

	Global placement						**Wirelength**						**Bin density**						**Cost gradient**					
	SEQ	**T=1**	**T=2**	**T=4**	**T=8**	**T=12**	**SEQ**	**T=1**	**T=2**	**T=4**	**T=8**	**T=12**	**SEQ**	**T=1**	**T=2**	**T=4**	**T=8**	**T=12**	**SEQ**	**T=1**	**T=2**	**T=4**	**T=8**	**T=12**
ADAPTEC1	1.82	1.85	2.28	2.76	3.05	3.09	1.76	1.87	2.53	3.01	3.26	3.34	5.39	4.48	5.33	7.31	9.27	9.21	2.22	2.39	3.58	5.10	6.58	7.17
ADAPTEC2	1.56	1.57	1.85	2.25	2.51	2.54	1.67	1.78	2.41	2.78	3.14	3.13	4.36	3.56	4.29	6.65	8.67	8.41	1.96	2.11	2.97	4.53	6.11	6.73
ADAPTEC3	1.50	1.49	1.92	2.47	2.93	3.07	1.63	1.74	2.27	2.60	2.83	2.86	3.19	2.74	3.88	6.63	9.45	9.99	1.73	1.85	2.63	3.85	5.46	6.44
ADAPTEC4	1.44	1.42	1.85	2.46	3.01	3.19	1.51	1.61	1.95	2.23	2.40	2.44	2.88	2.50	3.75	6.50	9.73	10.68	1.64	1.76	2.51	3.70	5.52	6.53
BIGBLUE1	1.93	1.90	2.44	2.95	3.26	3.37	1.64	1.72	2.27	2.63	2.92	3.06	5.56	4.48	5.93	9.42	11.50	11.99	2.37	2.50	3.76	5.35	6.61	7.08
BIGBLUE2	1.46	1.46	1.86	2.44	2.85	3.00	1.54	1.64	2.17	2.52	2.70	2.75	3.01	2.63	3.60	6.16	9.07	9.76	1.73	1.85	2.61	3.91	5.18	6.08
BIGBLUE4	1.57	1.47	1.84	2.45	3.00	3.22	1.81	1.79	2.50	3.07	3.41	3.45	3.16	2.53	3.26	5.71	8.72	9.96	1.81	1.82	2.45	3.61	5.26	6.42
geomean	**1.60**	**1.58**	**1.99**	**2.53**	**2.94**	**3.06**	**1.65**	**1.73**	**2.29**	**2.68**	**2.93**	**2.99**	**3.80**	**3.18**	**4.20**	**6.83**	**9.45**	**9.95**	**1.91**	**2.02**	**2.89**	**4.25**	**5.79**	**6.63**

result, sublinear performance improvements are reported as the number of threads increases.

The original multithreaded RePlAce achieves speedups on average ≈50% lower than ours. Multithreaded DREAMPlace is slower than single-threaded RePlAce as it currently lacks a multithreaded implementation of its forward and backward propagation passes [5].

VII. CONCLUSION

This paper proposes strategies to accelerate the RePlAce VLSI placer using off-the-shelf multi-core CPU hardware. We introduce cache-fitting data structures and explore variable privatization, loop fission, and per-thread workload balancing. When using 2–12 threads, these techniques accelerate bin density calculation by a factor of 4.2–10×, wirelength computation by a factor of 2.3–3×, and cost gradient calculation by a factor of 2.9–6.6×. As all these functions are inherently memory bound, sublinear speedups were expected. Future work will aim to improve data locality further and to investigate numerical stability issues.

ACKNOWLEDGMENTS

We thank the authors of RePlAce [3] for sharing their code.

REFERENCES

[1] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure*. NL: Springer, 2011.

[2] 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 Trans. on Design Automation of El. Systems (TODAES)*, vol. 20, no. 2, pp. A:1–A:33, Feb. 2015.

[3] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "RePlAce: Advancing solution quality and routability validation in global placement," *IEEE Trans. on CAD of Int. Circuits and Systems*, pp. 1–14, Jul. 2018.

[4] Y. Nesterov, "A method for solving a convex programming problem with convergence rate $O(1/k^2)$," *Soviet Mathematics Doklady*, vol. 27, no. 2, pp. 372–376, 1983.

[5] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *Proc. of the 56th Design Automation Conference*, Las Vegas, NV, USA, Jun. 2019, pp. 1–6.

[6] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," Nov. 2015.

[7] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "ISPD 2005 placement contest and benchmark suite," in *ISPD*, San Francisco, CA, USA, Apr. 2006, pp. 216–220.

[8] T. Lin, C. Chu, and G. Wu, "POLAR 3.0: An ultrafast global placement engine," in *Proc. of the 2015 IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, Austin, TX, USA, Nov 2015, pp. 520–527.

[9] C.-X. Lin and M. D. Wong, "Accelerate analytical placement with GPU: A generic approach," in *Proc. of the Design, Automation and Test in Europe Conf. and Ex.*, Dresden, Germany, Mar. 2018, pp. 1345–1350.

[10] A. Al-Kawam and H. M. Harmanani, "A parallel GPU implementation of the Timber Wolf placement algorithm," in *2015 12th International Conference on Information Technology - New Generations*, Las Vegas, NV, USA, Apr. 2015, pp. 792–795.

[11] B. Bredthauer, M. Olbrich, and E. Barke, "STP - a quadratic VLSI placement tool using graphic processing units," in *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, Geneva, Switzerland, Jun. 2018, pp. 77–84.

[12] I. Korovin, M. Khisamutdinov, G. Schaefer, and A. Kalyaev, "An efficient evolutionary approach to placing VLSI elements based on multithread CUDA programming," in *5th Intl. Conf. on Informatics, Electronics and Vision*, Dhaka, Bangladesh, May 2016, pp. 1153–1156.

[13] P. Oikonomou, M. G. Koziri, A. N. Dadaliaris, T. Loukopoulos, and G. I. Stamoulis, "Domocus: Lock free parallel legalization in standard cell placement," in *2017 6th Intl. Conf. on Modern Circuits and Systems Technologies (MOCAST)*, Thessaloniki, Greece, May 2017, pp. 1–4.

[14] M.-C. Kim, D.-J. Lee, and I. L. Markov, "SimPL: An effective placement algorithm," *IEEE Trans. on CAD of Int. Circuits and Systems*, vol. 31, no. 1, pp. 50–60, Jan. 2012.

[15] 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 Trans. on CAD of Int. Circuits and Systems*, vol. 27, no. 7, pp. 1228–1240, Jun. 2008.

[16] M.-K. Hsu, Y.-W. Chang, and V. Balabanov, "TSV-aware analytical placement for 3D IC designs," in *Proc. of the 48th Design Automation Conference*, San Diego, CA, USA, Jun. 2011, pp. 664–669.

[17] W. C. Naylor, R. Donelly, and L. Sha, "Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer," US Patent 6301693, 2001.