Accelerate Analytical Placement with GPU: A Generic Approach

Chun-Xun Lin* and Martin D. F. Wong†
*clin99@illinois.edu, †mdfwong@illinois.edu
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

***Abstract*—This paper presents a generic approach of exploiting GPU parallelism to speed up the essential computations in VLSI nonlinear analytical placement. We consider the computation of wirelength and density which are widely used as cost and constraint in nonlinear analytical placement. For wirelength gradient computing, we utilize the sparse characteristic of circuit graph to transform the compute-intensive portions into sparse matrix multiplications, which effectively optimizes the memory access pattern and mitigates the imbalance workload. For density, we introduce a computation flattening technique to achieve load balancing among threads and a High-Precision representation is integrated into our approach to guarantee the reproducibility. We have evaluated our method on a set of contest benchmarks from industry. The experimental results demonstrate our GPU method achieves a better performance over both the CPU methods and the straightforward GPU implementation.**

I. Introduction

VLSI global placement is a pivotal stage in physical design flow. A substantial amount of research efforts have been devoted on the global placement [1–10]. Among existing placement methods, the analytical approaches, especially the nonlinear placement, has obtained the best quality up to the present. However, compared with other approaches such as simulated annealing or partitioning, the nonlinear methods suffer from a slower performance. The reason is that nonlinear methods apply mathematical programming to derive the solution, which involves a huge numbers of arithmetic operations and becomes the bottleneck of performance. Therefore, the goal of this paper is to exploit the parallelism of GPU to speed up the computations in nonlinear placement.

GPU is well-known for its capability to conduct massive computations concurrently. There are several researches on applying GPU to EDA applications [11] [12] [13] [14] [15], and some focus on using GPU on EDA placement. The authors [11] propose a fast sparse matrix-vector multiplication method based on GPU and utilize the method to expedite a quadratic placer. Another paper [12] demonstrates the feasibility of accelerating simulated annealing placement with GPU. The placement models in both papers are different from state-of-the-art nonlinear placement and it is not clear how the methods can be extended to benefit nonlinear placement. The paper [13] applies GPU to optimize the performance of an analytical placer. The authors adopt a straightforward parallelization method such as delegating the outer loop of a nested loop to GPU threads which does not require modifying the original computing scheme. However, adherence to the CPU computing scheme refrains the method from fully exploiting potential parallelism brought by GPU and some critical issues such as imbalance workload cannot be effectively resolved due to the framework's inherent limitation. Besides, due to the limited GPU compute capability, their method has to compromise with reduced numerical accuracy which degrades the solution quality.

In this paper, we consider the cost model that is broadly used by existing nonlinear placement approaches. The cost model of nonlinear placement approaches can be generally formulated as

$$\begin{aligned} &\text{minimize} && \text{Wirelength} \\ &\text{subject to} && D_{bin} \leq D_{threshold} \end{aligned}$$

The wirelength is a differentiable function, e.g. log-sum-exp [16] or weighted average [17] that approximates the half-perimeter wirelength (HPWL) and the D_{bin} is the bin density on the layout. Mathematical optimization such as the iterative gradient descent method is commonly applied to minimize the cost. As a result, fast computation of the wirelength gradient and the bin density is important to the performance of the placer and two GPU approaches are developed to respectively accelerate the computing of the wirelength gradient and the density. We summarize our contributions as follows:

- Our method is faster than the CPU methods and can obtain further speedup over a straightforward GPU parallelization. The efficiency of our methods has been evaluated through experimenting on a set of contest benchmarks.
- Our method does not design for a specific placer, instead placers that adopt the same cost model can apply the proposed method to achieve performance improvement.
- Reproducibility is guaranteed in the proposed methods. A stable and reproducible output is particularly useful in software debugging.

II. Wirelength Computation

Wirelength is one of the most important cost functions in VLSI placement. A commonly used wirelength model is the half perimeter wirelength (HPWL) which sums the width and height of the bounding box formed by the pins. However, HPWL is a non-differentiable function and thus cannot be directly used in the analytical placement method. Several approximation models are proposed and one popular approach is the Logarithm-Sum-Exponential (LSE) model [16]. The LSE of a given net n with m pins on it can be calculated as follows:

$$\begin{aligned} LSE(n) = \gamma\{\ln(\sum_{i=0}^{m} e^{x_i/\gamma}) + \ln(\sum_{i=0}^{m} e^{-x_i/\gamma})\}+ \\ \gamma\{\ln(\sum_{i=0}^{m} e^{y_i/\gamma}) + \ln(\sum_{i=0}^{m} e^{-y_i/\gamma})\} \end{aligned} \quad (1)$$

978-3-9819263-0-9/DATE18/©2018 EDAA

In equation (1), each (x_i, y_i) is the x and y coordinate of a pin on n and γ is a predefined constant. Since the gradient calculation is identical in both x and y directions, we only discuss the computation on x below. For a pin connected with k nets, its wirelength gradient can be derived by differentiating the LSE equations of the k nets and sum them up. Below is the equation to calculate the wirelength gradient of a given pin p in the x direction (the gradient in the y direction can be derived using the same formula by replacing the x coordinate with the y coordinate).

$$Grad_x(p) = \sum_{i=0}^{k} \{ \frac{e^{x_p/\gamma}}{\sum_{v \in n_i} e^{v_x/\gamma}} \} - \sum_{i=0}^{k} \{ \frac{e^{-x_p/\gamma}}{\sum_{v \in n_i} e^{-v_x/\gamma}} \} \quad (2)$$

The implementation to calculate equation (2) on CPU can be divided into two steps. In the first step, for each net, we compute the summation of the exponential term for each pin on the net. In the second step, we use the summations from the first step following the equation 2 to derive the gradients of each pin in both x and y directions. Algorithm 1 shows the pseudo code to compute the wirelength gradient of each pin on the x coordinate, where the first step is from line 3 to line 12 and line 13 to 21 is the second step.

Algorithm 1 Wirelength gradient on x using CPU

Input: P : Pins, N : nets;
Output: $Grad$: gradients of each pin
1: $ExpSum \leftarrow \{\}$;
2: $NegExpSum \leftarrow \{\}$;
3: **for each** n in N **do**
4: $sum \leftarrow 0$;
5: $neg\_sum \leftarrow 0$;
6: **for each** p of n **do**
7: $sum \leftarrow sum + e^{x_p/\gamma}$;
8: $neg\_sum \leftarrow neg\_sum + e^{-x_p/\gamma}$;
9: **end for**
10: $ExpSum \leftarrow ExpSum \cup \{sum\}$;
11: $NegExpSum \leftarrow NegExpSum \cup \{neg\_sum\}$;
12: **end for**
13: **for each** p in P **do**
14: $left\_term \leftarrow 0$
15: $right\_term \leftarrow 0$
16: **for each** n of p **do**
17: $left\_term \leftarrow left\_term + 1/ExpSum[n]$
18: $right\_term \leftarrow right\_term + 1/NegExpSum[n]$
19: **end for**
20: $Grad[p] \leftarrow e^{x_p/\gamma} * left\_term - e^{-x_p/\gamma} * right\_term$
21: **end for**

A. Wirelength gradient on GPU

GPU is suitable for the gradient computation due to its ability to do massive computations concurrently. To utilize GPU for wirelength gradient computing, an intuitive way is to launch two kernels sequentially with the first kernel executing the first step and another for the second step. To be more specific, in the first kernel, we assign a thread to a net to compute the exponential sum of its pin coordinates, and in the second kernel, a thread is delegated to compute the gradients for a pin.

This method is simple and does not require any modification to the CPU algorithm. However, there are two deficiencies in this method:

- As the number of pins on nets are disparate and pins have different numbers of connected nets, the memory access can be very inefficient and threads can suffer from imbalance workload, for example, a pin coordinate can be read multiple times in different threads and some threads can perform more computations than others.
- Same values can be computed several times in different threads, leading to the inefficient use of computing resources. For example, a pin p_1 can be connected to nets n_1 and n_2 and thus the exponential value of the p_1 coordinate will be computed twice in different threads of the first kernel.

Algorithm 2 GPU Kernel 1 on exponential sum

Input: P : Pins, N : nets, $ExpSum$, $NegExpSum$;
1: $id \leftarrow blockSize * blockId + threadId$
2: $sum \leftarrow 0$;
3: $neg\_sum \leftarrow 0$;
4: **for each** p of $N[id]$ **do**
5: $sum \leftarrow sum + e^{x_p/\gamma}$;
6: $neg\_sum \leftarrow neg\_sum + e^{-x_p/\gamma}$;
7: **end for**
8: $ExpSum[id] \leftarrow \{sum\}$;
9: $NegExpSum[id] \leftarrow \{neg\_sum\}$;

Algorithm 3 GPU Kernel 2 on wirelength gradient

Input: P : Pins, N : nets, $ExpSum$, $NegExpSum$;
Output: $Grad$: gradients of each pin
1: $id \leftarrow blockSize * blockId + threadId$
2: $left\_term \leftarrow 0$;
3: $right\_term \leftarrow 0$;
4: **for each** n of $P[id]$ **do**
5: $left\_term \leftarrow left\_term + 1/ExpSum[n]$
6: $right\_term \leftarrow right\_term + 1/NegExpSum[n]$
7: **end for**
8: $Grad[id] \leftarrow e^{x_p/\gamma} * left\_term - e^{-x_p/\gamma} * right\_term$

B. Our GPU implementation

To overcome these deficiencies, we propose a new GPU implementation flow containing five steps. To prevent computing the same value repetitively among threads, we first launch a kernel to calculate the exponential values of each pin's coordinates and store the result in a vector for later use (Algorithm 4). Based on the fact that a circuit can be represented as a sparse graph, we construct a sparse $(0, 1)$-matrix where the rows correspond to nets, columns correspond to pins, and the value of an entry (i, j) is 1 if pin j is in net i. With the sparse matrix and the vector of the exponential values, the exponential sum of each net can be derived through multiplying the sparse matrix with the vector (Algorithm 5). The next step is to launch a kernel to compute the reciprocal of the exponential sum for each net (Algorithm 6). To calculate the summation of reciprocals for each pin, a kernel is used to sum the reciprocals of all connected nets and a key observation here is that the summation can also be obtained by multiplying a sparse matrix with the reciprocals (Algorithm 7), where the sparse matrix is the transpose of the sparse matrix in the second step. The last step is to derive the gradient by adding the sum of reciprocals for each pin.

A major concern of using GPU is the overhead incurred from data transfer between CPU and GPU. The proposed GPU method requires two data transfers, one is to transfer the pin coordinates from CPU to GPU memory in the beginning and another is to copy the gradients back to CPU memory.

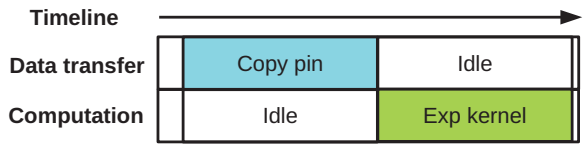
Algorithm 4 GPU Step 1 on exponential values

Input: P : Pins
Output: $ExpVal$, $NegExpVal$
1: $id \leftarrow blockSize * blockId + threadId$
2: $p \leftarrow P[id]$
3: $ExpVal[id] \leftarrow e^{x_p/\gamma}$
4: $NegExpVal[id] \leftarrow -e^{-x_p/\gamma}$

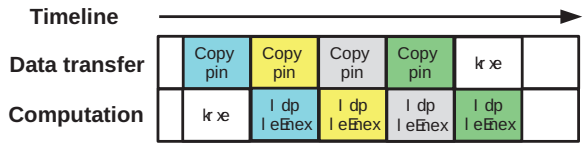
Algorithm 5 GPU Step 2 on exponential sum

Input: $ExpVal$, $NegExpVal$, P : Pins, N : Nets
Output: $ExpSum$, $NegExpSum$
1: $ExpSum \leftarrow SparseMatrix(P, N) * ExpVal$
2: $NegExpSum \leftarrow SparseMatrix(P, N) * NegExpVal$

Although the data transfer overhead is inevitable, we can further reduce the overhead by using streams. A stream is similar to a job queue which holds GPU operations to be executed sequentially, whereas operations in separate streams can run concurrently if available resource exists. Hence, we can create several streams to overlap data transfers with computations through dispatching GPU operations on subsets of data to different streams. Considering overlapping the step one kernel (Algorithm 4) by copying pin coordinates to GPU, we first divide pins into disjoint subsets and map each subset to a stream, then a copy operation and a kernel for computing exponential value are enqueued into each stream to operate on the corresponding data. By having multiple streams process different subsets, we can keep the copy device and execution device occupied [18] as shown in Figure 1.



(a) Data transfer without overlapped with computations.



(b) Overlap data transfer with computations through streams. Operations with the same color are enqueued in the same stream.

Fig. 1: Comparison of data transfer with and without using streams

Our proposed flow has two benefits over the straightforward GPU implementation:

- We transform the two nested loops, the most time-consuming parts, to two sparse matrix multiplications. A sparse matrix can be stored in various formats such as a compressed sparse row (csr) or a coordinate list (coo) and those data structures unleash more opportunities to optimize the memory access and reorder the computations for balancing the workload.
- Data movement between processing units is a common bottleneck in heterogeneous computing and our approach reduces the overhead by overlapping computations with data transfers through utilizing streams.

Algorithm 6 GPU Step 3 on reciprocal exponential values

Input: $ExpSum$, $NegExpSum$
Output: $RecExpVal$, $RecNegExpVal$
1: $id \leftarrow blockSize * blockId + threadId$
2: $RecExpVal[id] \leftarrow 1/ExpSum[id]$
3: $RecNegExpVal[id] \leftarrow 1/NegExpSum[id]$

Algorithm 7 GPU Step 4 on summation of reciprocals

Input: $RecExpSum$, $RecNegExpSum$, P, N
Output: $RecSum$, $RecNegSum$
1: $RecSum \leftarrow SparseMatrix(P, N)^T * RecExpSum$
2: $RecNegSum \leftarrow SparseMatrix(P, N)^T * RecNegExpSum$

Algorithm 8 GPU Step 5 on gradient of each pin

Input: $RecSum$, $RecNegSum$, $ExpVal$, $NegExpVal$, P, N
Output: $Grad$
1: $id \leftarrow blockSize * blockId + threadId$
2: $Grad[id] \leftarrow RecSum[id] * ExpVal[id] +$
3: $\quad RecNegSum[id] * NegExpVal[id]$

III. Density Computation

Density computation is an essential step in analytical placement methods. During the optimization process, the density will be evaluated in every iteration and the placement can stop once the cells' overlap is lower than a predefined threshold. In this section, we first formulate the density computation problem and present a CPU implementation, then we demonstrate a straightforward GPU implementation and discuss its deficiencies. Lastly, we propose a High-Precision GPU implementation and introduce two techniques to further improve the performance.

A. Density problem formulation

We consider a general formulation where the layout is a two-dimensional grid and the cells C to be placed are rectangular. To compute the density, the first step is to accumulate the overlapped area between bins and the cells. Then the density can be derived by dividing the accumulated area in each bin with unit bin area (a coarser density map can be formed by combining multiple bins into a single bin). Parallelizing the second step is pretty straightforward, so in this paper we will focus on parallelization of the overlapped area accumulation.

B. Area accumulation on CPU

Algorithm 9 Area accumulation on CPU

Input: $Cells$, $Grid$
Output: $OverlapArea$
1: **for each** $b \in Grid$ **do**
2: $\quad OverlapArea[b] \leftarrow 0$
3: **end for**
4: **for each** $c \in C$ **do**
5: $\quad overlap\_bins \leftarrow FindOverlapBins(Grid, c)$
6: $\quad$ **for each** $b \in overlap\_bins$ **do**
7: $\quad\quad area \leftarrow FinOverlapArea(b, c)$
8: $\quad\quad OverlapArea[b] \leftarrow area + OverlapArea[b]$
9: $\quad$ **end for**
10: **end for**

To compute the overlapped area between cells and bins, an intuitive way is to loop through each cell and add the overlapped area of the cell to corresponding bins. Algorithm 9 is the pseudo code to accumulate overlapped area for each bin.

Reproducibility is guaranteed in this CPU implementation as the floating point additions are executed in deterministic order.

To speed up the computation, an instinctive way is to use the multiple cores in CPU to have several threads doing the accumulation concurrently. Consider line 4 in Algorithm 9, cells can be partitioned into subsets with nearly equal size and each thread is responsible for a subset of cells. A challenge of parallel programming is to maintain the data integrity under multi-thread execution. Notice that in line 8, the accumulated area in each bin is shared among all threads and concurrent access to this data might result in data race.

A simple and efficient solution is to use atomic operation. An atomic operation serializes the access to the data without using locks, which protects the data from running into the race condition and can still maintain good performance. However, in C++ the standard library does not provide atomic operations for the floating type. An alternative way is to implement the atomic floating operation via the atomic *compare-and-exchange* instruction. The compare-and-exchange instruction atomically checks whether the destination value is equal to a given value and replaces the destination value by a new value if the predicate is satisfied. Therefore, a thread can first take a snapshot of the destination bin and then use the compare-and-exchange instruction with the snapshot value to update the bin. Algorithm 10 presents the pseudo code of the multi-threaded area accumulation. In lines 1 and 2, we launch multiple threads and assign a subset of cells to each thread. From lines 3 to 14, each thread adds the overlapped area of each cell to the corresponding bins via the compare-and-exchange instruction in line 11.

Algorithm 10 Multi-threaded area accumulation on CPU

Input: $Cells$, $Grid$
Output: Bin
1: launchThreads()
2: $myCell \leftarrow AssignCells(myThreadId, Cells)$;
3: **for each** $c \in myCell$ **do**
4: $overlap\_bins \leftarrow FindOverlapBins(Grid, c)$
5: **for each** $b \in overlap\_bins$ **do**
6: $area \leftarrow FindOverlapArea(b, c)$
7: $update \leftarrow false$
8: **while** $update \neq true$ **do**
9: $snapshot \leftarrow Bin[b]$
10: $new\_value \leftarrow snapshot + area$
11: $update \leftarrow CAE(Bin[b], snapshot, new\_value)$
12: **end while**
13: **end for**
14: **end for**
15: synchronizeAllThreads()

C. Area accumulation on GPU

The multi-threaded CPU approach of area accumulation can also be applied to GPU. For the GPU method, we launch a kernel with assigning a thread to each cell to compute the overlapped area among bins. In contrast to the CPU, modern GPUs support atomic operation for the floating type, circumventing the need of using compare-and-exchange instruction. However, there are two deficiencies in this approach:

- No guarantee of reproducibility: Although the atomic operation resolves the data race problem, it does not admit reproducibility. The reason is that floating point arithmetic is non-associative [19]. As atomic operation does not enforce a deterministic order on thread execution, given same operands the result could be slightly different every time.
- Imbalanced workload: As the size of the cells is not uniform, the number of bins cross by different cell could differ greatly. Therefore, divergence might occur in the kernel due to the inconsistent iterations among threads (line 6 in Algorithm 9), which might hamper the performance.

To conquer the first problem, we adopt a High-Precision method [20]. The method represents every floating number by N 64 bits integers where each integer carries a fraction of the floating number and N controls the representable range of floating number. Our idea is to perform accumulation on the integers converted from floating numbers and the resulting integers can then be translated into floating numbers after the accumulation finishes. As the arithmetic addition on integers is associative, given the same inputs the outcome of accumulation will be identical. Another benefit of this method is obviating the need to impose a predefined execution order among threads which might be detrimental to performance.

To evenly distribute the workload among threads, we come up with a computation flattening technique. Since the dimensions of cells and bins are known in the beginning and remain unchanged during placement, we can derive the maximum number of bins that intersect with each cell before placement. With this information, we can determine the total number of threads to be launched by summing up the number of bins intersecting with each cell and assign a thread to compute the overlap area between a cell and one of its intersected bin and add the result to the corresponding bin. For example, Figure 2 shows flattening computation of two cells.

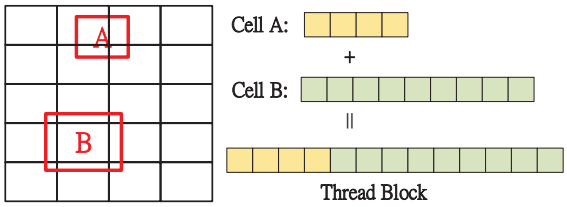


Fig. 2: An example illustrates computation flattening. Cells A and B intersect with four and nine bins respectively and thirteen threads are created to compute the overlapped area for each portion.

This approach also requires two data transfers: one is to send the cell coordinates to GPU memory and another is to fetch the accumulated area from GPU memory. To further reduce the transfer overhead, we use stream to overlap both data transfers with two computation kernels. For the cell coordinates transfer, we map subsets of cells to streams and a kernel is enqueued into each stream to compute the overlapped area in the High-Precision format for the cells. For the second data transfer, we associate subset of bins to streams and each stream enqueues a kernel to convert the integers to floating numbers for the bins and copy the results to CPU memory. Algorithm 11 is the pseudo code of the High-Precision GPU method with streams.

Algorithm 11 High-Precision area accumulation on GPU

Input: $Cells$, $Grid$, $myCellId$, $myBinId$, $numStream$
Output: Bin
1: $cudaMemSet(hpBinArea, 0)$
2: $cudaCreateStream(Streams, numStream)$;
3: **for each** $s \in Streams$ **do**
4: $s \leftarrow copyPin(myCellId, Cells)$
5: $s \leftarrow countOverlap(myCellId, Cells, hpBinArea)$
6: **end for**
7: $cudaDeviceSynochronize()$
8: **for each** $s \in Streams$ **do**
9: $s \leftarrow convertToFloat(myBinId, hpBinArea, fBinArea)$
10: $s \leftarrow copyDensity(myBinId, fBinArea, Bin)$
11: **end for**

IV. Experimental Results

We implement all programs in C++. The GPU used in the experiment is NVIDIA GeForce GTX 1080 and the CPU is Xeon 3.0 GHz Quad cores with 32 GB memory. We implement a gradient descent placer based on the LSE wirelength model and conduct experiments on the benchmarks from the 2015 ISPD routing-driven placement contest [21]. The statistic of the benchmarks is in Table I. For GPU programs, we record the runtime from host (CPU) side, including kernel launch latency, the overhead of pin coordinates (host to GPU) and results (GPU to host) transfer.

TABLE I: Benchmark Statistic

Benchmark	Cells	Nets
mgc_fft_1	32,281	33,307
mgc_fft_2	32,281	33,307
mgc_matrix_mult_1	155,325	158,527
mgc_matrix_mult_a	149,650	154,284
mgc_matrix_mult_b	146,435	151,612
mgc_pci_bridge32_a	29,517	29,985
mgc_pci_bridge32_b	28,914	29,417
mgc_des_perf_1	112,644	112,878
mgc_des_perf_a	108,288	110,281
mgc_des_perf_b	112,644	112,878
mgc_edit_dist_a	127,413	131,134
mgc_fft_a	30,625	32,088
mgc_fft_b	30,625	32,088
mgc_superblue12	1,286,948	1,293,413
mgc_superblue11_a	925,616	935,613
mgc_superblue16_a	680,450	697,303

A. Wirelength

We implement the wirelength gradient computation with four methods: CPU, CPU with four threads, straightforward GPU parallelization, and our proposed GPU method. The cuSPARSE library [22] is adopted for sparse matrix multiplication in our method. The wirelength gradient computation is not affected by cells' locations, and we report the results in one iteration. Table II lists the runtime of the four methods. Among the four methods, the CPU method is the slowest and the proposed GPU method is the fastest over all test cases. Consider the average speedup, our method outperforms the CPU, CPU with four threads and the GPU loop parallelization by 173×, 93× and 8× respectively.

B. Density

For density experiment, we implement four methods for comparison: CPU, CPU with four threads (with the compare-and-exchange technique to ensure data integrity), proposed GPU method with and without using streams. We set the N in the High-Precision method to 3, i.e., each floating number is represented by three 64-bits integers.

For each test case, the placement stops when successive cell displacement is small and we record the average computation time. The grid size is 2048×2048 for the superblue family and 1024×1024 for the others. Table III lists the runtime for each test case. From the Table III, the GPU method with streams has the best performance in all test cases while the multi-threaded CPU method only obtains minor improvement in few benchmarks. The compare-and-exchange operation is the primary cause for the slower performance of multi-threaded method. Unlike the atomic operation which serializes the access to data, the threads that failed to update the bin using the compare-and-exchange operation have to retrieve the new value and compete for the access (lines 8-12 in Algorithm 10), resulting in high overhead when there are many overlaps between cells.

V. Conclusion

In this paper, we present GPU approaches to accelerate the wirelength gradient and density computation. For the wirelength gradient computing, we convert the summations into sparse matrix multiplications, which effectively mitigates the non-uniform workload among threads and increases the performance. For the density computing, we propose a computation flattening technique to completely resolve the imbalance workload. With the CUDA stream, runtime can be further reduced by overlapping the computation and data transfer. Lastly, a High-Precision method is integrated into our approach to ensure reproducible results.

VI. ACKNOWLEDGMENT

This work is partially supported by the National Science Foundation under Grant CCF-1421563 and CCF-171883.

References

[1] A.B. Kahng and Qinke Wang. Implementation and extensibility of an analytic placer. *IEEE Trans. on CAD*, 24(5):734–747, May 2005.
[2] Carl Sechen and Alberto Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *IEEE Journal of Solid-State Circuits*, 20(2):510–522, 1985.
[3] Mehmet Can Yildiz and Patrick H Madden. Improved cut sequences for partitioning based placement. In *Proc. ACM/IEEE DAC*, pages 776–779. ACM, 2001.
[4] N. Viswanathan, Min Pan, and C. Chu. Fastplace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. In *Proc. IEEE/ACM ASP-DAC*, pages 135–140, 2007.
[5] Tung-Chieh Chen, Tien-Chang Hsu, Zhe-Wei Jiang, and Yao-Wen Chang. NTUplace: A ratio partitioning based placement algorithm for large-scale mixed-size designs. In *Proc. ACM ISPD*, pages 236–238, 2005.
[6] Tony Chan, Jason Cong, and Kenton Sze. Multilevel generalized force-directed method for circuit placement. In *Proc. ACM ISPD*, pages 185–192, 2005.
[7] P. Spindler and F.M. Johannes. Fast and robust quadratic placement combined with an exact linear net model. In *Proc. IEEE/ACM ICCAD*, pages 179–186, Nov 2006.
[8] Ulrich Brenner, Markus Struzyna, and Jens Vygen. Bonnplace: Placement of leading-edge chips by advanced combinatorial algorithms. *IEEE Trans. on CAD*, 27(9), 2008.

TABLE II: Wirelength gradient computation (μs)

Benchmark	CPU (A)	CPU mt (B)	GPU (C)	Our (D)	A/D	B/D	C/D
mgc fft 1	62818	42856	3965	922	68.13	46.48	4.30
mgc fft 2	63155	45405	3962	878	71.93	51.71	4.51
mgc matrix mult 1	250341	146679	7145	1074	233.09	136.57	6.65
mgc matrix mult a	241860	160679	6883	1050	230.34	153.03	6.56
mgc matrix mult b	235409	153353	6824	1028	229.00	149.18	6.64
mgc pci bridge32 a	58918	36293	5889	645	91.35	56.27	9.13
mgc pci bridge32 b	57155	34299	5828	600	95.26	57.17	9.71
mgc des perf 1	174971	95828	16798	865	202.28	110.78	19.42
mgc des perf a	168382	95718	15887	852	197.63	112.35	18.65
mgc des perf b	174674	105847	17087	884	197.60	119.74	19.33
mgc edit dist a	205182	116110	11107	991	207.05	117.16	11.21
mgc fft a	61156	35510	3742	876	69.81	40.54	4.27
mgc fft b	61032	34818	3767	896	68.12	38.86	4.20
mgc superblue12	2520856	933960	38512	10083	250.01	92.63	3.82
mgc superblue11 a	1619743	636320	16169	5671	285.62	112.21	2.85
mgc superblue16 a	1194524	464355	12480	4301	277.73	107.96	2.90
Avg speedup					173.43	93.91	8.38

TABLE III: Area accumulation (μs)

Benchmark	CPU (A)	CPU mt (B)	GPU w/o stream (C)	GPU w/ stream (D)	A/D	B/D	C/D
mgc_fft_1	11290.84	12493.41	7321.55	2419.29	4.67	5.16	3.03
mgc_fft_2	8093.70	8860.68	6047.33	2189.98	3.70	4.05	2.76
mgc_matrix_mult_1	15005.26	19605.24	7917.75	3059.32	4.90	6.41	2.59
mgc_matrix_mult_a	6846.51	8750.37	4906.24	2083.73	3.29	4.20	2.35
mgc_matrix_mult_b	7027.02	8671.27	4974.14	2117.35	3.32	4.10	2.35
mgc_pci_bridge32_a	6003.96	5876.86	5507.50	2078.25	2.89	2.83	2.65
mgc_pci_bridge32_b	3173.07	2993.10	4425.70	1794.05	1.77	1.67	2.47
mgc_des_perf_1	12641.25	18864.96	7431.84	2556.22	4.95	7.38	2.91
mgc_des_perf_a	6966.61	8736.17	5253.17	2084.12	3.34	4.19	2.52
mgc_des_perf_b	9942.77	13533.10	6257.44	2408.73	4.13	5.62	2.60
mgc_edit_dist_a	9579.94	12900.03	6292.26	2613.12	3.67	4.94	2.41
mgc_fft_a	3158.54	3378.08	4385.56	1790.72	1.76	1.89	2.45
mgc_fft_b	3011.20	3355.41	4348.91	1778.45	1.69	1.89	2.45
mgc_superblue12	80574.98	73570.18	26326.61	11367.5	7.09	6.47	2.32
mgc_superblue11_a	49883.19	57774.44	19044.40	8728.10	5.72	6.62	2.18
mgc_superblue16_a	32067.20	29805.87	16426.19	7477.87	4.29	3.99	2.20

[9] Myung-Chul Kim, Dong-Jin Lee, and I.L. Markov. SimPL: An effective placement algorithm. *IEEE Trans. on CAD*, 31(1):50–60, Jan 2012.

[10] Jingwei Lu, Pengwen Chen, Chin-Chih Chang, Lu Sha, Dennis J-.H. Huang, Chin-Chi Teng, and Chung-Kuan Cheng. ePlace: Electrostatics based placement using Nesterov's method. In *Proc. ACM/IEEE DAC*, pages 121:1–121:6, 2014.

[11] Y. Deng, B. D. Wang, and S. Mu. Taming irregular EDA applications on GPUs. In *Proc. IEEE/ACM ICCAD*, pages 539–546, Nov 2009.

[12] A. Al-Kawam and H. M. Harmanani. A parallel GPU implementation of the TimberWolf placement algorithm. In *2015 12th International Conference on Information Technology - New Generations*, pages 792–795, April 2015.

[13] Jason Cong and Yi Zou. Parallel multi-level analytical global placement on graphics processing units. In *Proc. IEEE/ACM ICCAD*, pages 681–688, 2009.

[14] K. Zhai, W. Yu, and H. Zhuang. GPU-friendly floating random walk algorithm for capacitance extraction of VLSI interconnects. In *DATE*, pages 1661–1666, March 2013.

[15] Y. Han, K. Chakraborty, and S. Roy. A global router on GPU architecture. In *ICCD*, pages 78–84, Oct 2013.

[16] 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, October 9 2001. US Patent 6,301,693.

[17] M. K. Hsu, Y. W. Chang, and V. Balabanov. TSV-aware analytical placement for 3D IC designs. In *Proc. ACM/IEEE DAC*, pages 664–669, June 2011.

[18] How to overlap data transfers in CUDA c/c++: https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/.

[19] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[20] Patrick E Small, Rajiv K Kalia, Aiichiro Nakano, and Priya Vashishta. Order-invariant real number summation: Circumventing accuracy loss for multimillion summands on multiple parallel architectures. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 152–160. IEEE, 2016.

[21] Ismail S. Bustany, David Chinnery, Joseph R. Shinnerl, and Vladimir Yutsis. ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement. In *Proc ACM ISPD*, pages 157–164, 2015.

[22] cuSPARSE library. https://developer.nvidia.com/cusparse.