GPU-Accelerated Rectilinear Steiner Tree Generation

Zizheng Guo[1†], Feng Gu[2†], Yibo Lin[1,3*]
[1]School of Integrated Circuits, Peking University [2]School of Computer Science, Peking University
[3]Beijing Advanced Innovation Center for Integrated Circuits, Beijing, China
{gzz, gu_feng, yibolin}@pku.edu.cn

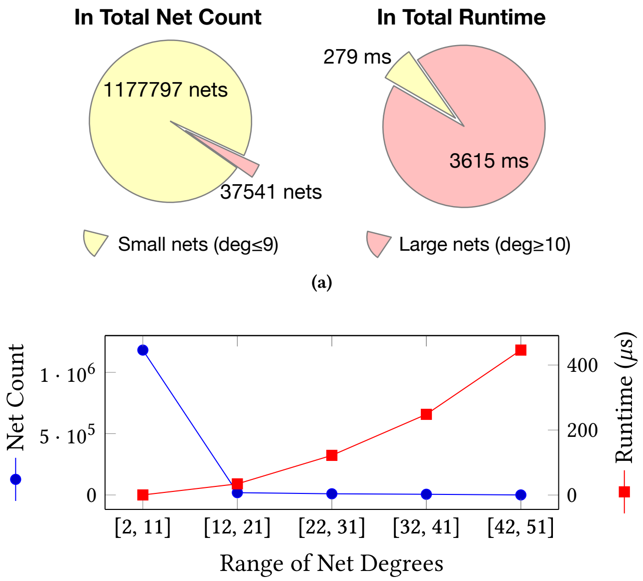
ABSTRACT

Rectilinear Steiner minimum tree (RSMT) generation is a fundamental component in the VLSI design automation flow. Due to its extensive usage in circuit design iterations at early design stages like synthesis, placement, and routing, the performance of RSMT generation is critical for a reasonable design turnaround time. State-of-the-art RSMT generation algorithms, like fast look-up table estimation (FLUTE), are constrained by CPU-based parallelism with limited runtime improvements. The acceleration of RSMT on GPUs is an important yet difficult task, due to the complex and non-trivial divide-and-conquer computation patterns with recursions. In this paper, we present the first GPU-accelerated RSMT generation algorithm based on FLUTE. By designing GPU-efficient data structures and levelized decomposition, table look-up, and merging operations, we incorporate large-scale data parallelism into the generation of Steiner trees. An up to 10.47× runtime speed-up has been achieved compared with FLUTE running on 40 CPU cores, filling in a critical missing component in today's GPU-accelerated design automation framework.

1 INTRODUCTION

The construction of rectilinear Steiner minimum tree (RSMT) is an important problem in VLSI design automation. RSMT gives the smallest tree with rectilinear wires that connect all given pins of a net on the circuit layout. It provides insight into various critical circuit design metrics, such as net routability, capacitive load, and timing. As such, RSMT generation is frequently used in early circuit design stages like synthesis, placement, and routing, where the circuit design space is explored to optimize the chip quality. During these stages, RSMT is evaluated within the inner loop of extensive optimization iterations. Being executed thousands of times on large circuits with millions of nets and pins, the speed of RSMT generation is critical for a reasonable design turnaround time.

Due to the NP-completeness of RSMT generation [1], current research generally works on the trade-off between runtime and solution quality [2–10]. Better RSMT solution quality calls for better heuristics, at the cost of longer runtime. Among them, the current most efficient and widely-adopted heuristic is FLUTE [9], which stands for fast look-up table estimation. FLUTE precomputes a look-up table for nets with degree ≤ 9, for which the best solutions can be quickly determined. For larger nets, FLUTE tries to break them down into smaller nets which are solved recursively and merged back to the solution of the original ones. Other similar directions work on the rectilinear tree with shortest-path, called rectilinear Steiner minimum arborescence (RSMA) [11–15], or combining the objective of RSMT and RSMA, called shallow-light tree (SLT) [16–21].



(a)

(b)

Figure 1: (a) Statistics of high-degree and low-degree nets in benchmark superblue1 and their portions in overall FLUTE runtime. (b) Distribution of net degrees and the average FLUTE runtime per net on CPU.

Despite all these efforts on improving the quality and speed of RSMT generation, the performance demand increases even faster. Current RSMT generation algorithms are all constrained with CPU-based parallelism, which provides limited runtime improvements given both more design iterations and larger design sizes. Although most of the nets in a typical circuit design have only a small degree (≤ 9), larger nets are exponentially harder to solve and, in fact, take up most of the RSMT runtime, as shown in Figure 1.

To speed up the VLSI design flow, GPUs have been incorporated in various design stages, such as logic simulation [22], placement [23], timing analysis [24], and routing [25, 26]. However, GPU acceleration is extremely challenging for the RSMT generation problem. Current RSMT algorithms, such as FLUTE, are based on a divide-and-conquer strategy with deep recursions, which are impossible to be executed on GPU threads with very limited stack memory. The sizes of nets in a circuit netlist are highly uneven, from 2-pin nets to nets with 40 pins or more, which leads to an extremely imbalanced workload and harms the parallelism. The computation patterns and branching behaviors are determined at runtime by the specific shapes of nets, which makes it hard to allocate GPU memory and causes GPU thread divergence.

Therefore, in this paper, we propose a GPU-accelerated RSMT generation algorithm, which, to our best knowledge, is the first one in literature. Our algorithm is built on top of FLUTE heuristics, with

[†] First two authors contributed equally. [*] Corresponding author

ICCAD '22, October 30-November 3, 2022, Gainesville, FL, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549434

highly-efficient GPU kernels offloading critical RSMT computations to large-scale GPU parallelism. We summarize four major contributions of this work as follows:

- We propose a levelized task decomposition strategy applicable to all kinds of circuit net size distributions, which ensures a balanced workload and enables high-performance data parallelism for RSMT generation on GPUs.
- We eliminate the recursion patterns of FLUTE by careful algorithmic transforms. We have solved the central computational challenges for GPU-accelerated RSMT generation.
- We design GPU-efficient kernels for all RSMT generation operations, including net decomposition, table look-up, and solution merging, which produce Steiner trees that are as good as FLUTE, but at a much faster speed.
- By accelerating RSMT on GPUs, we fill in a critical missing component in today's GPU-accelerated design automation framework. Both our algorithms and our methodology can bring performance benefits to many different design stages.

We evaluate our algorithm on large-scale industrial designs from ICCAD 2015 timing-driven placement contest [27]. Our algorithm produces RSMT solutions that are identical to FLUTE, but much faster. We have achieved an average 29.47× speed-up compared to FLUTE running on a single CPU, and 10.47× compared to FLUTE on 40 CPUs. We have also explored the effect of accuracy parameters on performance and demonstrated the superior scalability of our algorithm. Our GPU-accelerated algorithm can compute RSMT solutions with a higher accuracy setting given the same runtime constraint. To the extreme, our algorithm with a low-accuracy setting can even outperform the runtime of half-perimeter wirelength (HPWL) on CPU.

The rest of this paper is organized as follows. Section 2 introduces the background of RSMT generation. Section 3 presents details of our GPU-based RSMT generation algorithm based on FLUTE. Section 4 demonstrated the experimental results of our algorithm. Finally, Section 5 concludes the paper.

2 RECTILINEAR STEINER MINIMUM TREE

RSMT generation is a central problem in VLSI design automation, formulated as follows: *Given a set of nets with pins on a two-dimensional (2D) plane, construct horizontal and vertical (i.e. rectilinear) wires for each net that connect all the pins and possible Steiner points, with minimum total wirelength.* The wirelength is minimized to reduce routing resource usage, signal delay, as well as capacitive load of the nets. The rectilinear constraint comes from the VLSI technology where interconnects are set up on Manhattan grids. The RSMT solutions, as well as the total wirelength in terms of Steiner trees, serve as an important indicator of circuit performance and are widely used in the VLSI design automation framework. For example, in timing-driven cell placement stage [27], RSMT is used to generate rough routing results and timing-related parasitics information during placement iterations.

To better describe the RSMT generation problem, Hanan grids [28] are introduced that discretize the search space of RSMTs by drawing one horizontal line and one vertical line through each node, as shown in Figure 2(b). A net with n pins prior to RSMT generation is then described as three arrays $xs[0...n-1]$, $ys[0...n-1]$, and $s[0...n-1]$. The first two arrays denote the discretized and sorted x and y coordinates which form the grid of all possible horizontal and vertical wires. The last array, s, locates all pins on this grid by providing a permutation of $\{0, 1, ..., n-1\}$, as shown in Figure 2(c). Specifically, $s[i]$ gives the rank of x-coordinate at the pin with the i-th smallest y-coordinate. It was proven in [28] that this formulation, forcing all horizontal and vertical wires to reside on the grid, retains the best RSMT solutions with minimal wirelength. The RSMTs are equivalently constructed on the Hanan grid of each net in the rest of this work.

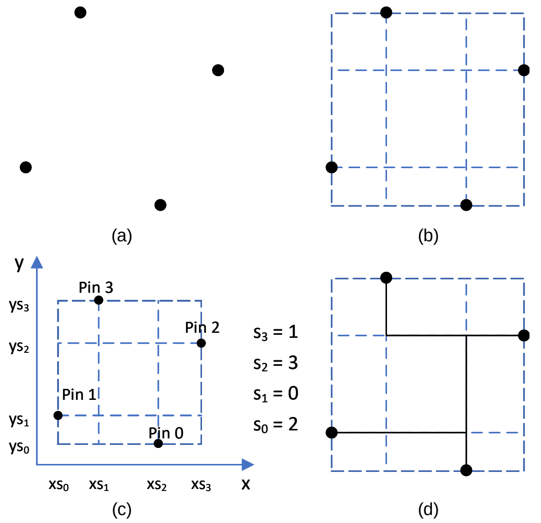


Figure 2: The Hanan-grid model of a net with degree 4. (a) A net of degree 4. (b) The Hanan-grid of the net. (c) The xs, ys and s of the net. (d) A possible Steiner tree of the net.

2.1 Heuristics of RSMT Generation

There have long been research works on the trade-off between RSMT generation runtime and solution quality, due to the NP-completeness and intractable nature of exact RSMT generation [1]. Exact RSMT solutions, such as GeoSteiner [2, 3], require up to exponential runtime which is not feasible for VLSI applications. One line of research works on a cheap alternative to RSMT, called rectilinear minimum spanning tree (R-MST). R-MST does not insert any Steiner points and can be efficiently constructed in $O(n \log n)$ time for a net with n pins [4], but at the cost of up to 50% worse result than RSMT [5]. Kahng *et al* [8] propose to iteratively improve on top of an initial R-MST solution by contracting edges. A modest average of 11% improvement has been obtained. Better RSMT solution quality calls for better heuristics, at the cost of longer runtime [6, 7]. A recent work by Liu *et al* [10] applies machine learning techniques, especially reinforcement learning (RL) and attention mechanism, to RSMT generation.

Among these research works, the current most efficient and widely-adopted heuristic is FLUTE [9], which stands for fast look-up table estimation. FLUTE computes RSMT by breaking large nets into smaller ones, whose solutions are recursively computed and merged back to the RSMT of the original nets, as illustrated in Figure 3. For small nets (with degree≤ 9), FLUTE solves them using a precomputed look-up table that guarantees the best quality and fast speed. The look-up table is organized as sets of possible RSMT solutions indexed by the Hanan grid permutation array $s[0, ..., n-1]$, which has about $9! = 362880$ different possibilities. For breaking large nets, FLUTE tries to find the possible *breaking pins* using a weighted sum of a few different heuristics and explores them under the control of an accuracy parameter $\mathcal{A}$. At merging, all pairs of subnets are back connected by their breaking pins and evaluated to keep the solution with the smallest wirelength.

Figure 4 shows the sequential FLUTE algorithm running on CPU. Before calling FLUTE, the nets are transformed to Hanan grids as described before. First, the entry point of FLUTE accepts the net data consisting of Hanan grid arrays xs, ys, s, and the required accuracy

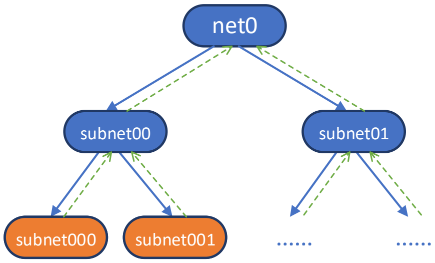


Figure 3: Dataflow of recursive divide-and-conquer-based RSMT generation, such as FLUTE. Blue arrows indicate recursive algorithm calls on divided nets, and green dashed arrows indicate returns at merging.

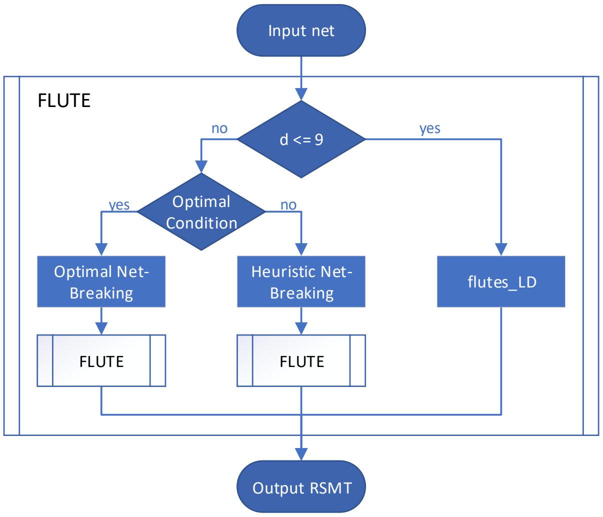


Figure 4: Flow chart of the sequential FLUTE algorithm, for parallelism explanation purposes.

$\mathcal{A}$. When the degree of input net falls under the range of look-up tables, FLUTE matches the net against look-up tables for solutions and computes the optimal RSMT directly (denoted as `flutes_LD`). Otherwise, the control flow passes to large degree handling logics, where the net breaking strategies are explored. Specifically, the net is either broken into exactly two subnets when there is a provably optimal net breaking pin separating the search space, or up to $\mathcal{A}$ different net breaking pins are tried subsequently leading to $2\mathcal{A}$ subnets in case no optimality is guaranteed.

2.2 GPU Architecture and Challenges

Heterogeneous computing systems consisting of CPUs and GPUs, as shown in Figure 5, have brought huge performance benefits to all kinds of scientific computing applications, thanks to the architectural difference between CPUs and GPUs that caters their advantages in different types of workloads. CPUs have a few large and powerful cores, with high performance single-thread computation as well as branching prediction capability. This makes CPUs suitable for general purpose computation tasks with complex control structures. Multi-threading on CPUs is also promising provided with a limited number of concurrent tasks. Given massively parallel tasks consisting of possibly thousands to even millions of threads, CPUs incur high multitasking and thread switching overhead, as well as low data bandwidth and cache storage. On the contrary, GPU architecture is designed from the ground to fit the need of massive parallelism. A GPU has hundreds to thousands of small cores running in parallel. These cores can switch between threads with almost no runtime cost, thus effectively hiding memory latency when one group of threads is waiting for a memory request to complete.

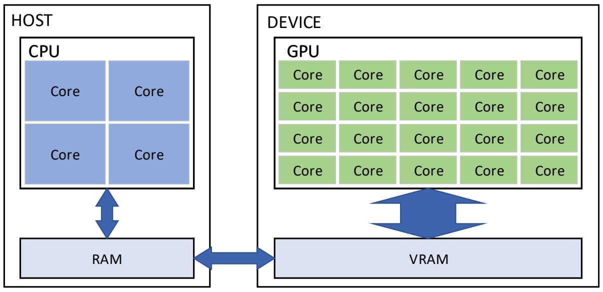


Figure 5: Heterogeneous computing system. The double-headed arrows indicate the bandwidth between two components.

However, GPUs are not suitable for every computation tasks. Specifically, GPU prefers large-scale parallelized simple tasks that have both similar computational patterns and workload. While this is the case for machine-learning (ML) tasks that are essentially matrix-vector multiplications, RSMT generation works differently and challenges arise as follows:

(1) Current RSMT algorithms, such as FLUTE, are based on a divide-and-conquer strategy with deep recursions. This type of program is impossible to be executed on GPU threads which have very limited stack memory.
(2) The degrees of nets in a circuit netlist are highly uneven, from 2-pin nets to nets with 40 pins or more. Worse still, the amount of computation increases exponentially with the increase of degree. This leads to an extremely imbalanced workload and harms the parallelism.
(3) The computation patterns and branching behaviors are determined at runtime by the specific shape of pin distributions inside nets. This introduces GPU thread divergence and also makes it hard to allocate and manage GPU memory spaces.

As a result, accelerating RSMT on GPU requires us to not only eliminate the harmful recursive patterns inside current heuristics, but also explore high-quality, massive parallelism beyond the nets.

3 ALGORITHM

In this section, we explain the details of our GPU-accelerated RSMT generation algorithm. The overall task graph of our GPU-accelerated algorithm is shown in Figure 6, where the arrows indicate the data dependency between tasks. As shown in the figure, our algorithm can be divided into three main stages: *initialization*, *breaking*, and *merging*. We note that the break and merge stages work in an iterative way rather than the recursive mode in FLUTE, as shown by arrows pointing backward. There is no extensive data copy between CPU and GPU during the inner algorithm loops which ensures minimal overhead of CPU-GPU communication.

3.1 Parallelism Inside Subnets

In this section, we introduce the parallelism inside FLUTE recursions which we use in later algorithms.

The divide-and-conquer FLUTE process as described in Section 2, together with a deep dive into net breaking details, yields a new level of parallelism and the solution to eliminate workload imbalance. We draw an example net breaking step in Figure 7 which uses the same example in Figure 2(c)[1]. We break the example net at two positions. In each break, the breaking pin in the original net becomes a common

[1] In fact, this net is already a low-degree net. We break it just for explanation here.

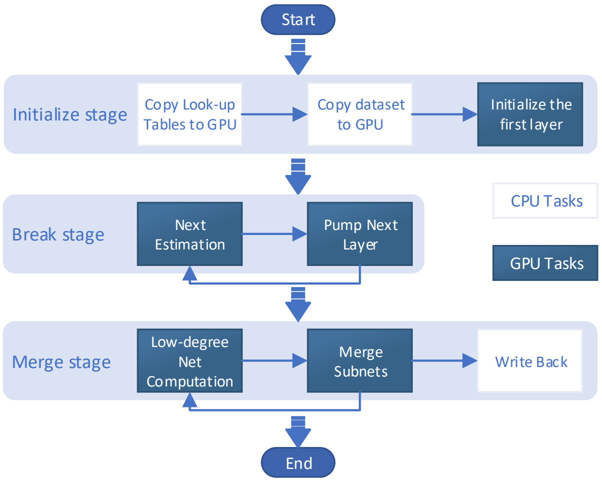


Figure 6: Overall task flow of our GPU-accelerated RSMT generation algorithm.

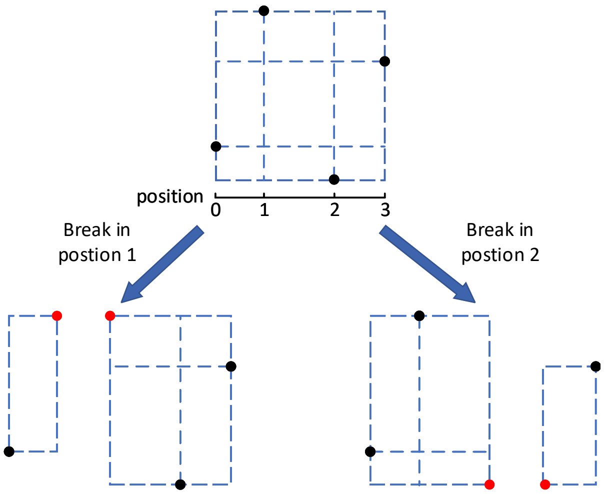


Figure 7: An example that breaks a net into subnets.

pin in the subnets (highlighted in red), and it later serves as a bridge to merge solutions of them. The common pin isolates the computation of the two subnets, making them independent of each other. The up to $\mathcal{A}$ of different breaking pins can also be determined in advance during net breaking. As large nets typically go through multiple passes of breaking and merging, a large number of intermediate subnets can be processed in parallel. In parallel CPU-based FLUTE, there are only 1M of original nets running concurrently. However, up to 100M intermediate subnets are generated during the breakings. Moreover, although the workload varies by net degrees, the amount of computation for breaking and merging steps remain constant, which is very helpful in GPU acceleration.

3.2 The Levelized Representation

This section introduces the basic data structure in our algorithm. This data structure arranges all intermediate subnets into layers indicating the recursion relations. Instead of the depth-first search (DFS) process in FLUTE, this data structure maintains a parallelized breadth-first search (BFS), as illustrated in Figure 8. While the breaking and merging relations of nets and subnets remain unchanged from Figure 3, this introduces a different order of execution and parallel strategy. The initial layer is filled with Hanan grids of input nets. In the breaking stage, nets in the same layer are broken in parallel to form the next layer. In the merging stage, nets are merged or queried in look-up tables in parallel from the last layer to the first layer.

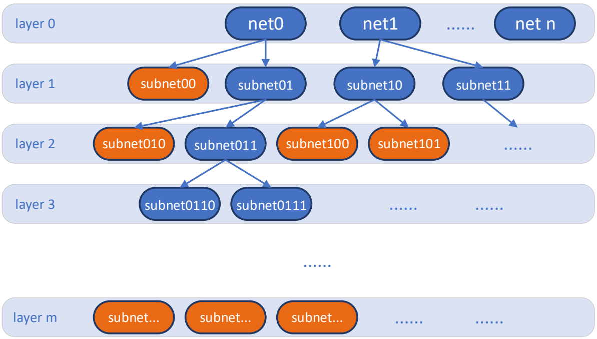


Figure 8: The levelized representation with subnets in the same layer running in parallel.

3.3 GPU-Accelerated Net Breaking

In this section, we present the GPU-accelerated net breaking algorithm. As shown in Algorithm 1, the central step is an iterative process to break nets into smaller ones level by level, until all nets become small enough to solve by FLUTE look-up tables.

Algorithm 1: GPU-Accelerated Breaking Stage

```
Input: $N$ as #nets, $L$ as #layers
Input: $levels[0..L][0..N]$, the buffer of each layer
Data: $next\_location[0..N]$, the location of the first subnet
Output: the subnets of each net
for $i = 0 \to \infty$ do
    $level \leftarrow levels[i]$;
    estimate_next_location($level, next\_location$);
    prefix_sum($next\_location, next\_location + N$);
    if all nets in this level are low-degree nets then break;
    $nxt\_level \leftarrow levels[i+1]$;
    break_kernel($level, nxt\_level, next\_location$);
end
```

There are two important procedures in this algorithm, which are called `estimate_next_location` and `break_kernel`. The first procedure preallocates memory for all parent nets to store its subnets in the next layer. There is one array to store subnet metadata, and another array to store the pin information. Different nets emit different numbers of subnets and pins, based on the net degree and pin distribution. The numbers of subnets and pins are collected by Algorithm 2 into two arrays, *next_index* and *next_offset*, on which we launch parallel *prefix sum* operations, as shown in Figure 9. We note that the prefix sum is efficient on GPU using Thrust library [29]. The prefix sum yields the total amount of memory to allocate, and the offset to which the subnets should be stored.

After preparing for the memory offset, we break the nets using Algorithm 3. The net breaking heuristics are the same as the sequential version of FLUTE introduced in Section 3.1. The key difference is that we do not wait for the subnets to complete RSMT computation or merge them at this time. Instead, we leave these tasks later when we stop at low-degree nets and merge back layer by layer.

3.4 GPU-Accelerated Net Merging

In the previous stage, we begin with the first layer and break the nets iteratively until the last layer in which there are no high-degree nets.

Algorithm 2: Estimate Next Location

```
Input: $N$ as #nets
Input: $d$, the degree of the net
Input: $net\_acc[0..N]$, the accuracy of the nets
Output: $next\_index[0..N]$, the index of the first subnet
Output: $next\_offset[0..N]$, the offset of the first subnet
/* Process one net w/ blockDim.x threads
    */
$netID$ = blockIdx.x * blockDim.x + threadIdx.x;
$acc \leftarrow net\_acc[netID]$;
$newacc \leftarrow$ update $acc$;
if $d \leq 9$ then
    $nxns = 0$;
    $nxpn = 0$;
end
else if Optimal Net-Breaking then
    $nxns = 2$;
    $nxpn = d + 2$;
end
else
    $nxns = newacc * 2$;
    $nxpn = newacc * (d + 1)$;
end
$next\_index[netID + 1] = nxns$;
$next\_offset[netID + 1] = nxpn$;
if $netID == 0$ then
    $next\_index[0] = 0$;
    $next\_offset[0] = 0$;
end
```

Algorithm 3: Break kernel

```
Input: $N$ as #nets, $NN$ as #subnets
Input: $d$ as the degree, $acc$ as the accuracy
Input: $level[0..N]$, the buffer of the current layer
Input: $next\_location[0..N]$, the location of the first subnet
Output: $next\_level[0..NN]$, the buffer of the next layer
/* Process one net w/ blockDim.x threads
    */
$netID$ = blockIdx.x * blockDim.x + threadIdx.x;
$net \leftarrow level[netID]$;
$nxtl \leftarrow next\_location[netID]$;
if $d \leq 9$ then return;
else if Optimal Condition then
    Break $net$ optimally into $subnet0$, $subnet1$;
    $next\_level[nxtl] \leftarrow subnet0$;
    $next\_level[nxtl + 1] \leftarrow subnet1$;
end
else
    for $i = 0 \rightarrow acc - 1$ do
        Break $net$ heuristically into $subnet0$, $subnet1$;
        $next\_level[nxtl + 2 * i] \leftarrow subnet0$;
        $next\_level[nxtl + 2 * i + 1] \leftarrow subnet1$;
    end
end
```

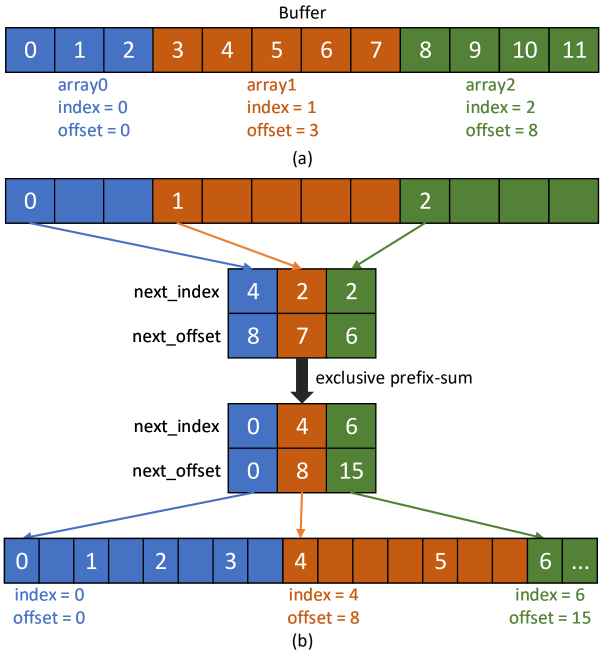


Figure 9: (a) An example of the index and offset of 3 arrays in a buffer. (b) Estimate the next location of arrays in (a). Array0 is broken heuristically into 4 subnets with a total degree of 8. Array1 is broken optimally into 2 subnets with a total degree of 7. The same happens to array2.

At this stage, what we need to do is an inverse process of the previous stage. We begin with the last layer, and keep merging the subnets into their parent nets and store them back in the previous layer. When we come to the first layer, we will get the RSMT of the initial nets. We show this process in Algorithm 4.

The workflow of Algorithm 4 is the same as the Algorithm 3. In the break stage, when we meet a low-degree net, we just leave it there and return with no subnets in the next layer. In this stage, we use `flutes_LD` to construct the RSMT of the low-degree nets, which is the same as the original CPU-based FLUTE.

Finally, we use a complete example to illustrate our algorithm in Figure 10. The break stage uses two kernels. The first kernel breaks net0 into four subnets, among which subnet03 is a low-degree net and the others are high-degree nets. The second kernel breaks the 3 high-degree subnets into a total of 6 subnets. Now all the subnets are low-degree nets; the break stage finishes. The merge stage uses three kernels. The first kernel constructs RSMTs for all the subnets in the last layer. The second kernel merges the six subnets into three RSMTs corresponding to subnet00, subnet01, and subnet02. Note that subnet03 bypasses layer2 and its RSMT is constructed in this kernel. The third kernel merges the four subnets into RSMT0 which is the RSMT of net0. In practice, millions of nets are computed in parallel inside the same levelized data structure.

4 EXPERIMENTAL RESULTS

We implemented our GPU-accelerated RSMT generation algorithm using C++ and CUDA and evaluated the results using ICCAD 2015 contest benchmarks [27]. The contest benchmarks contain 8 large industrial circuits, with placed netlists. These benchmarks were originally used as test cases for timing-driven gate placement engines. As a result, they provide realistic RSMT problem instances that an RSMT generation algorithm inside the inner loop of a timing-driven placement engine will encounter in practice. We compare our algorithm

Algorithm 4: Merge kernel

```
Input: $N$ as #nets, $NN$ as #subnets
Input: $d$ as the degree, $acc$ as the accuracy
Input: $next\_location[0..N]$, the location of the first subnet
Input: $level[0..N]$, the buffer of the current layer
Input: $next\_rsmt[0..N]$, the buffer of RSMT of the next layer
Output: $rsmt[0..N]$, the buffer of RSMT of the current layer
/* Process one net w/ blockDim.x threads
   */
$netID$ = blockIdx.x * blockDim.x + threadIdx.x;
$nxtl \leftarrow next\_location[netID]$;
if $d \leq 9$ then
    $rsmt[netID] \leftarrow$ flutes_LD($d$, $level[netID]$);
end
else if Optimal Condition then
    $rsmt[netID] \leftarrow$ merge_tree($next\_rsmt[nxtl]$,
      $next\_rsmt[nxtl + 1]$);
end
else
    for $i = 0 \rightarrow acc - 1$ do
        $t1 \leftarrow nxt\_rsmt\_level[nxtl + 2 * i]$;
        $t2 \leftarrow nxt\_rsmt\_level[nxtl + 2 * i + 1]$;
        Update $bestt1$ and $bestt2$ with $t1$ and $t2$;
    end
    $rsmt[netID] \leftarrow$ merge_tree($bestt1$, $bestt2$);
end
```

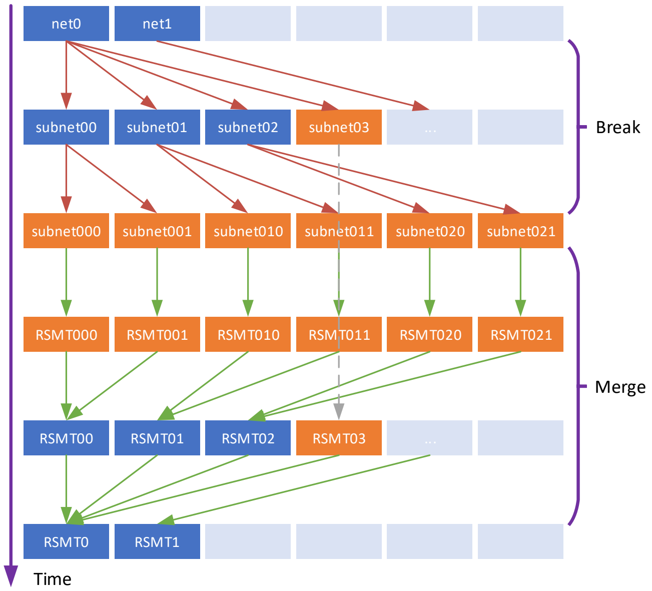


Figure 10: A complete example of levelized RSMT computation on GPU. The blue blocks indicate high-degree nets. The orange blocks indicate low-degree nets. The red arrows are operations in break kernel. The green arrows are operations in merge kernel.

with the original CPU-based FLUTE [9]. We note that FLUTE was also used as the golden standard in the ICCAD 2015 contest evaluation scripts. As the original FLUTE does not support multi-threading, we wrap it inside OpenMP threads to parallelize its computation on CPUs. We compile the programs using GNU GCC-7.5.0 and NVCC 11.5 and run them on a Ubuntu Linux machine with 40 Intel Xeon CPU cores at 2.10 GHz, 512GB memory, and 1 Nvidia A40 GPU with 48GB GPU memory and Nvidia Ampere architecture. All GPU kernels are run with 128 threads in each CUDA thread block, and $\lceil n/128 \rceil$ block to cover all n individual computation tasks (e.g., nets in the current level). Our algorithm is insensitive to thread allocation tweaks, thanks to the large amount of parallelism and the dynamic block dispatching mechanism inside the CUDA runtime. All runtime data is an average of 5 runs. We do not include the overhead for copying RSMT problem instances from CPU to GPU, and copying the resulting Steiner trees back to CPU, because they are not needed in any fully GPU-accelerated design automation framework [23, 30] where all data of pins and nets reside on GPU during the entire design process. We note that a CPU-based RSMT algorithm, e.g., FLUTE, will *instead* need to pay for the CPU-GPU copying overhead to fit into such frameworks [30].

4.1 Overall Comparison

Table 1 lists the benchmark statistics and the performance comparison between FLUTE [9] and our GPU-accelerated RSMT generation algorithm. We measure the runtime for completing 1 iteration of RSMT computation on all nets of the 8 ICCAD 2015 contest benchmarks, each consisting of about 1M nets and 3–6M pins. We set the accuracy parameter $\mathcal{A}$ to 8 (high accuracy) in both FLUTE and our algorithm, which is the default setting for evaluation in ICCAD 2015 contest. We do not show wirelength results in the table because our results are as good as (i.e. identical to) FLUTE.

According to the experiments, we have achieved a significant speedup across all benchmarks. It takes FLUTE 3689.8 ms to compute RSMT for the benchmark `superblue1` on one CPU, which means 3.6 seconds multiplied by the number of design iterations. Even with 40 CPU cores, FLUTE still needs 1232.0 ms to finish the RSMT generation. On the contrary, our algorithm needs only 116 ms to accomplish the same task on 1 GPU. This yields a 31.80× speed-up compared to FLUTE on 1 CPU, and 10.62× compared to the parallel version of FLUTE on 40 CPUs. On average, we have achieved 29.47× speed-up compared to FLUTE on 1 CPU, and 10.47× compared to FLUTE on 40 CPUs. The largest speed-up ratio happens on `superblue4`, where over 15× has been achieved compared to 40 CPUs. Remarkably, we are able to finish RSMT generation for every benchmark within 300 ms, which was an impossible task before this work. In a typical timing-driven gate placement task with 600–1000 iterations and RSMT generation in each iteration, we can save tens of minutes to even hours of design flow runtime. These results clearly demonstrate the efficiency of our GPU-accelerated RSMT algorithm.

4.2 Scalability Comparison

In this section, we compare our GPU-based algorithm with the scalability of CPU-based FLUTE on different numbers of CPU threads. Figure 11 shows the result on some representative benchmarks. The CPU-based FLUTE becomes faster with an increasing number of CPU threads from 1 to 16. The best speed-up compared to 1 CPU is around 3–4× with 16 CPU cores. However, no more performance benefit can be obtained when the number of CPU threads increases beyond 16. For some cases, FLUTE on 40 cores is even slightly slower than on 16 cores. These results point out the fundamental scalability limitations of CPU-based multi-thread parallelism, which was also reported on other different workloads [31–34]. On the other hand, our GPU-accelerated algorithm obtains superior performance with just a single GPU. There is a remarkable gap (about 10×) even compared with

Table 1: Performance comparison between FLUTE (running under both 1 CPU and 40 CPUs) and our GPU-accelerated algorithm (1 GPU) completing the RSMT generation tasks on ICCAD 2015 contest benchmarks.

Benchmark	Statistics					Runtime			Speed-up	
	#Nets	#Pins	Max Deg	Avg Deg	Tree Sum	1 CPU [9]	40 CPUs [9]	GPU (Ours)	1 CPU	40 CPUs
superblue1	1215338	3759909	51	3.094	5089142	3689.8 ms	1232.0 ms	116.0 ms	31.809×	10.621×
superblue3	1224360	3896306	51	3.182	5343892	4441.4 ms	1471.2 ms	138.6 ms	32.045×	10.615×
superblue4	802258	2488842	51	3.102	3373168	2771.4 ms	1381.4 ms	88.4 ms	31.351×	15.627×
superblue5	1097025	3237374	51	2.951	4280698	2881.0 ms	947.4 ms	91.2 ms	31.590×	10.388×
superblue7	1933378	6358017	51	3.289	8849278	8028.6 ms	2294.4 ms	275.4 ms	29.153×	8.331×
superblue10	1897768	5548092	51	2.923	7300648	5416.2 ms	1966.0 ms	203.4 ms	26.628×	9.666×
superblue16	999600	3005840	51	3.007	4012480	2833.4 ms	1213.8 ms	125.4 ms	22.595×	9.679×
superblue18	771244	2553668	51	3.311	3564848	3249.0 ms	939.4 ms	106.0 ms	30.651×	8.862×
Average			-			4163.8 ms	1430.7 ms	143.1 ms	29.477×	10.473×

#Nets: number of nets **#Pins**: number of pins **Max Deg**: max degree (number of pins) among all nets **Avg Deg**: averge net degree
Tree Sum: The total number of pins in RSMT results, including Steiner points.

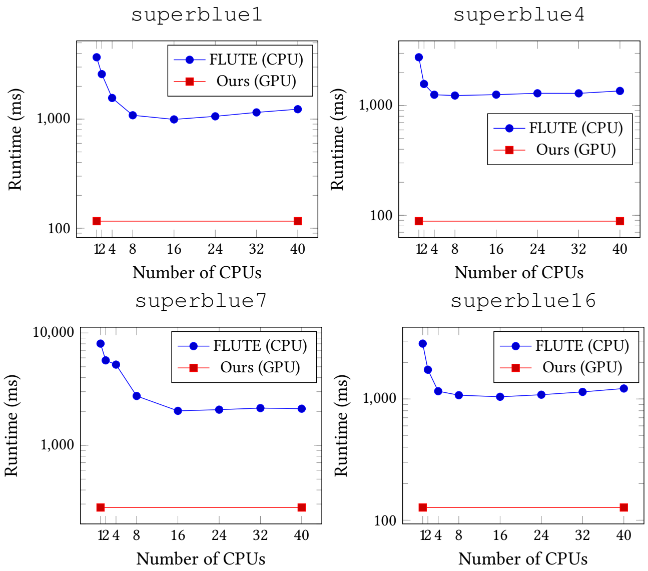


Figure 11: Runtime values at different numbers of threads. Accuracy is set to $\mathcal{A} = 8$.

the best CPU-based parallelism. Our work not only removes the obstacles to GPU-accelerated RSMT generation problem itself but also further justifies the effectiveness and necessity of parallelizing design automation on GPUs.

4.3 Effect of Accuracy Parameter

In this section, we study the effect of different accuracy parameters $\mathcal{A}$ on the performance of GPU acceleration. As introduced in Section 2, $\mathcal{A}$ controls the trade-off between accuracy and speed. Smaller $\mathcal{A}$ reduces the runtime at the cost of suboptimal results. FLUTE sets the default value of $\mathcal{A}$ as 3 which allows moderate error. The ICCAD 2015 contest sets $\mathcal{A}$ to 8 which produces very accurate results for most nets. As a result, we select $\mathcal{A}$ to be 1, 3, 6, 8, and 12 and compare the performance.

Figure 12 shows a detailed comparison. Both FLUTE and our algorithm incur larger runtime with larger accuracy, whereas our algorithm provides a large speed-up on all the five accuracy settings. Remarkably, our runtime to compute RSMT with $\mathcal{A} = 8$ is even faster than FLUTE with $\mathcal{A} = 1$, with much better solution quality. Generally, the speed-up ratio is larger for smaller accuracy settings. For example, the average speed-up for $\mathcal{A} = 1$ is 13.78× across all 8 benchmarks. This ratio is only slightly degraded to 9.77× with $\mathcal{A}$ set to our maximum value, 12. As a larger accuracy poses exponentially more demand on the recursive divide-and-conquer process, these results demonstrate the effectiveness of our levelized task decomposition strategy and algorithmic transforms eliminating recursion patterns, both making it possible to efficiently accelerate RSMT computation on GPUs.

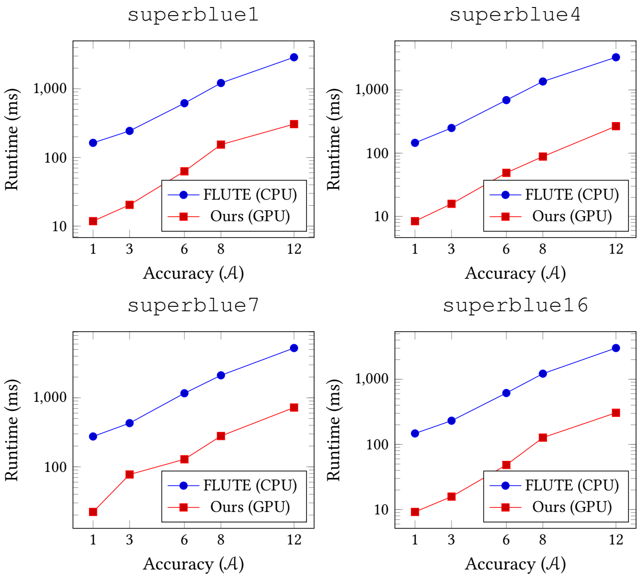


Figure 12: Runtime values at different accuracy parameters. FLUTE is run with 40 CPU cores.

4.4 Comparison with HPWL Runtime

In this section, we explore the speed and accuracy trade-off beyond RSMT generation. In VLSI design flow, HPWL is widely used as a fast approximation to net wirelength. HPWL tends to be overly optimistic as it does not guarantee a solution to net interconnects. While RSMT gives a more realistic wirelength approximation, HPWL is still widely used because it could be computed much faster. With our GPU-accelerated RSMT generation algorithm, however, the runtime of RSMT generation is improved significantly, bringing it even on par with HPWL. To show this, we run a detailed comparison between

HPWL, FLUTE, and our GPU-accelerated algorithm. The RSMT algorithms are run with accuracy setting $\mathcal{A} = 1$. Although this allows aggressive error tolerance, it is argued in [9] that this is still a better approximation than HPWL. We show the runtime results in Table 2. We can confirm that the original FLUTE is much slower than HPWL. Even when FLUTE is run on 40 CPUs, it costs 177.3 ms on average with the lowest accuracy setting. On the other hand, HPWL needs just 13.4 ms on a single CPU, and is even faster on 40 CPUs. With our GPU-accelerated RSMT generation algorithm, we have reduced the runtime of RSMT computation to an average of 13.1 ms, outperforming HPWL on a single CPU for the first time. The massive GPU parallelism not only speeds up the existing RSMT algorithms but also unlocks more scenarios for the applications of RSMT in the design flow which may in turn benefit the quality of circuit design. We note that our GPU-accelerated runtime is slower than HPWL running on 40 CPUs. In our future works, we hope to investigate further into reducing the communication and GPU kernel launching overhead, which constitute a large portion of our 13.1 ms runtime.

Table 2: The runtime of HPWL compared with RSMT generation algorithms, with accuracy set to $\mathcal{A} = 1$.

Benchmark	HPWL		FLUTE [9]	Ours
	1 CPU	40 CPUs	40 CPUs	GPU
superblue1	13.4 ms	4.6 ms	157.6 ms	12.0 ms
superblue3	13.2 ms	4.4 ms	178.0 ms	14.6 ms
superblue4	8.2 ms	3.4 ms	146.2 ms	8.6 ms
superblue5	12.0 ms	4.4 ms	134.6 ms	10.4 ms
superblue7	21.4 ms	7.6 ms	280.6 ms	22.2 ms
superblue10	20.4 ms	6.6 ms	236.2 ms	15.8 ms
superblue16	10.8 ms	3.8 ms	149.4 ms	9.4 ms
superblue18	8.2 ms	3.2 ms	136.0 ms	11.8 ms
Average	13.4 ms	4.7 ms	177.3 ms	13.1 ms

4.5 Effect of Net Degree

In this section, we take a deep dive into the performance of GPU acceleration under different ranges of net degrees. As we have shown in Figure 1, compared to smaller nets, large nets have very different RSMT computation patterns, due to their exponentially more complex divide-and-conquer process. As a result, larger nets are more prone to acceleration challenges. For each benchmark, we categorize all nets into five groups by their degree ranges: [2, 11], [12, 21], [22, 31], [32, 41], and [42, 51]. For each degree range, we then sample 10000 different nets from its group. The nets are replicated in case a group has an insufficient number of nets. Finally, we run our GPU-accelerated algorithm on top of all sampled groups, and measure the runtime against FLUTE with 40 CPUs. Our results are shown in Figure 13. Both FLUTE and our GPU-accelerated algorithm incur an exponential runtime increase with larger nets, which confirms our summary in Figure 1. However, with GPU-accelerated RSMT algorithm, we introduce performance benefits to all different degree ranges. Generally, smaller nets are more efficiently accelerated with >10× speed-up. Although the largest nets with degree≥42 are very difficult to parallelize, we have about 1.5–2× speed-up for them as well. While this experiment setting helps us isolate different degree ranges to test their performance, we note that it gives artifically balanced workload which is favorable to the original CPU-based FLUTE. Thanks to our levelized net representation, our algorithm can deal with imbalanced workload more effectively compared to FLUTE in a real scenario where different net degrees are mixed together, as shown in previous experiments where we can achieve > 10× speed-up on such scenarios.

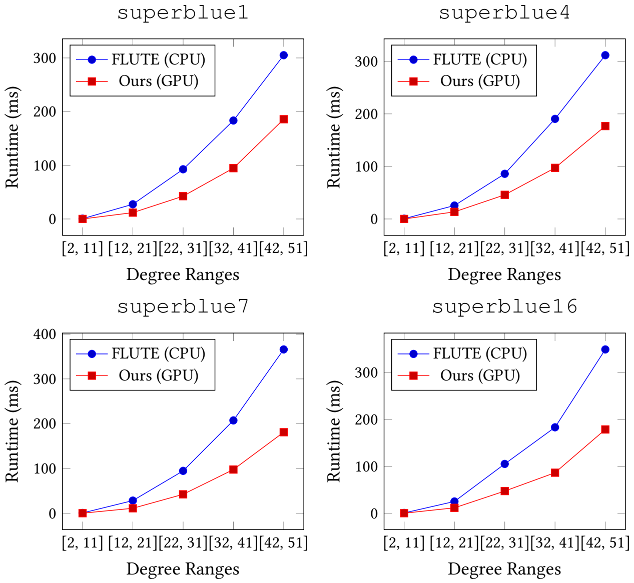


Figure 13: Runtime for computing RSMT of 10000 nets in different degree ranges. FLUTE is run with 40 CPU cores.

5 CONCLUSION

In this paper, we have proposed the first GPU-accelerated algorithm for RSMT generation. Built on top of FLUTE heuristics, we design highly-efficient GPU kernels offloading critical RSMT computations to large-scale GPU parallelism. We rewrite the algorithm to eliminate recursion patterns and propose a levelized task decomposition strategy applicable to all kinds of circuit net size distributions, all contributing to a balanced RSMT workload and high-performance data-parallel RSMT computation on GPUs. Our algorithm produces RSMT solutions that are as good as FLUTE, but much faster. An average of 29.47× speed-up has been achieved compared with FLUTE on a single CPU core, and 10.47× compared with FLUTE on 40 CPU cores. Given the same runtime constraint, we can also compute RSMT solutions with a higher accuracy setting. To the extreme, our algorithm with a low-accuracy setting can even outperform the runtime of half-perimeter wirelength (HPWL) on a single CPU. Our algorithm has filled in a critical missing component in today's GPU-accelerated VLSI design automation framework, with benefits to many different design stages. We plan to further optimize our algorithm by introducing CUDA graphs and better CPU-GPU scheduling, and integrate our algorithm into existing GPU-accelerated VLSI design automation frameworks like [23, 30].

ACKNOWLEDGE

This project is supported in part by the National Key Research and Development Program of China (No. 2021ZD0114702).

REFERENCES

[1] M. R. Garey and D. S. Johnson, “The rectilinear steiner tree problem is np-complete,” *SIAM Journal on Applied Mathematics*, vol. 32, no. 4, pp. 826–834, 1977.

[2] D. M. Warme, P. Winter, and M. Zachariasen, “Exact algorithms for plane steiner tree problems: A computational study,” in *Advances in Steiner trees.* Springer, 2000, pp. 81–116.

[3] GeoSteiner, Inc., “Geosteiner: Software for computing steiner trees,” http://www.geosteiner.com/, 2017.

[4] F. K. Hwang, “An o (n log n) algorithm for rectilinear minimal spanning trees,” *Journal of the ACM (JACM)*, vol. 26, no. 2, pp. 177–182, 1979.

[5] ——, “On steiner minimal trees with rectilinear distance,” *SIAM journal on Applied Mathematics*, vol. 30, no. 1, pp. 104–114, 1976.

[6] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang, “Closing the gap: Near-optimal steiner trees in polynomial time,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 11, pp. 1351–1365, 1994.

[7] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley, “A new heuristic for rectilinear steiner trees,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 10, pp. 1129–1139, 2000.

[8] A. B. Kahng, I. I. Măndoiu, and A. Z. Zelikovsky, “Highly scalable algorithms for rectilinear and octilinear steiner trees,” in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, 2003, pp. 827–833.

[9] C. Chu and Y.-C. Wong, “FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for VLSI design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 1, pp. 70–83, 2008.

[10] J. Liu, G. Chen, and E. F. Young, “Rest: Constructing rectilinear steiner minimum tree via reinforcement learning,” in *2021 58th ACM/IEEE Design Automation Conference (DAC).* IEEE, 2021, pp. 1135–1140.

[11] W. Shi and C. Su, “The rectilinear steiner arborescence problem is np-complete,” *SIAM Journal on Computing*, vol. 35, no. 3, pp. 729–740, 2005.

[12] A. B. Kahng and G. Robins, “A new class of iterative steiner tree heuristics with good performance,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 11, no. 7, pp. 893–902, 1992.

[13] J. Córdova and Y.-H. Lee, “A heuristic algorithm for the rectilinear steiner arborescence problem,” in *Engineering Optimization.* Citeseer, 1994.

[14] J. Cong, K.-S. Leung, and D. Zhou, “Performance-driven interconnect design based on distributed rc delay model,” in *30th ACM/IEEE Design Automation Conference.* IEEE, 1993, pp. 606–611.

[15] M. Pan, C. Chu, and P. Patra, “A novel performance-driven topology design algorithm,” in *2007 Asia and South Pacific Design Automation Conference.* IEEE, 2007, pp. 244–249.

[16] G. Chen and E. F. Young, “Salt: provably good routing topology by a novel steiner shallow-light tree algorithm,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1217–1230, 2019.

[17] B. Awerbuch, A. Baratz, and D. Peleg, *Efficient broadcast and light-weight spanners.* Technical Report, 1992.

[18] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh, and C.-K. Wong, “Provably good performance-driven global routing,” *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 11, no. 6, pp. 739–752, 1992.

[19] S. Khuller, B. Raghavachari, and N. Young, “Balancing minimum spanning trees and shortest-path trees,” *Algorithmica*, vol. 14, no. 4, pp. 305–321, 1995.

[20] C. J. Alpert, T. C. Hu, J.-H. Huang, A. B. Kahng, and D. Karger, “Prim-dijkstra tradeoffs for improved performance-driven routing tree design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 7, pp. 890–896, 1995.

[21] C. J. Alpert, W.-K. Chow, K. Han, A. B. Kahng, Z. Li, D. Liu, and S. Venkatesh, “Prim-dijkstra revisited: Achieving superior timing-driven routing trees,” in *Proceedings of the 2018 International Symposium on Physical Design*, 2018, pp. 10–17.

[22] Y. Zhang, H. Ren, and B. Khailany, “Opportunities for rtl and gate level simulation using gpus (invited talk),” in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–5.

[23] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020.

[24] Z. Guo, T.-W. Huang, and Y. Lin, “Gpu-accelerated static timing analysis,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* ACM, 2020.

[25] S. Lin, J. Liu, and M. D. Wong, “Gamer: Gpu accelerated maze routing,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–8.

[26] S. Liu, P. Liao, Z. Chen, W. Lv, Y. Lin, and B. Yu, “Fastgr: Global routing on cpu-gpu with heterogeneous task graph scheduler,” in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, Antwerp, Belgium, March 2022.

[27] M. Kim, J. Hu, J. Li, and N. Viswanathan, “ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 921–926.

[28] M. Hanan, “On steiner’s problem with rectilinear distance,” *SIAM Journal on Applied Mathematics (SIAP)*, vol. 14, no. 2, pp. 255–265, 1966.

[29] N. Bell and J. Hoberock, “Thrust: A productivity-oriented library for cuda,” in *GPU computing gems Jade edition.* Elsevier, 2012, pp. 359–371.

[30] Z. Guo and Y. Lin, “Differentiable-timing-driven global placement,” in *ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1–6.

[31] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, “NCTU-GR 2.0: multithreaded collision-aware global routing with bounded-length maze routing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 32, no. 5, pp. 709–722, 2013.

[32] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, “OpenTimer v2: A New Parallel Incremental Timing Analysis Engine,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp. 1–1, 2021.

[33] Z. Guo, T.-W. Huang, and Y. Lin, “A provably good and practically efficient algorithm for common path pessimism removal in large designs,” in *ACM/IEEE Design Automation Conference (DAC).* ACM, 2021.

[34] H. Yang, K. Fung, Y. Zhao, Y. Lin, and B. Yu, “Mixed-cell-height legalization on cpu-gpu heterogeneous systems,” in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2022.