A Simple, Fast, and GPU-friendly Steiner-Tree Heuristic

Alex Fallin
Department of Computer Science
Texas State University
San Marcos, USA
waf13@txstate.edu

Aarti Kothari
Department of Computer Science
Texas State University
San Marcos, USA
a_k299@txstate.edu

Jiayuan He
Department of Computer Science
University of Texas at Austin
Austin, USA
hejy@cs.utexas.edu

Christopher Yanez
Department of Computer Science
Texas State University
San Marcos, USA
cmy18@txstate.edu

Keshav Pingali
Department of Computer Science
University of Texas at Austin
Austin, USA
pingali@cs.utexas.edu

Rajit Manohar
Department of Computer Science
Yale University
New Haven, USA
rajit.manohar@yale.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, USA
burtscher@txstate.edu

Rectilinear-Steiner-Minimum-Tree (RSMT) generation is a key step in VLSI routing. This paper describes a deterministic RSMT heuristic named SFP because it is simple, fast, and parallel. It is based on a rapid method for computing the optimal solution for three-pin nets, which it extends and applies to larger nets. Compared to FLUTE, a fast and high-quality RSMT algorithm, SFP incurs a wirelength penalty of 0.38 percent on the 16 ISPD 2008 inputs with 13 million nets. However, on a 16-core 2950X CPU, parallel SFP is 19 times faster than FLUTE, and on a Titan V GPU, it is 86 times faster. Serial SFP can generate an RSMT for a 10,000-pin net in under 1.5 seconds.

Keywords—GPU acceleration, Parallelism, Routing, RSMT heuristic, Steiner points

I. Introduction

Signal routing is an essential step in the physical design of a VLSI chip. Its goal is to determine where to run the wires that connect the logic blocks (e.g., macros, cells, and gates) while keeping certain factors like wirelength, signal-propagation time, and power consumption low. This study focuses on the Rectilinear Steiner-Minimum-Tree generation—i.e., the first step of signal routing that involves minimizing the overall wirelength. The input is the location of the pins of each logic block as well as information about which pins to connect. A set of pins that must be connected is called a net. All nets together are referred to as the netlist.

Panel (a) in Fig. 1 provides an example of a net with three pins that connects the output (P1) of one logic gate to one input each (P2 and P3) of two other logic gates. The three pins can be connected as shown in panel (b) but doing so results in an unnecessarily long wirelength. Panel (c) shows a better solution that is optimal assuming we can only connect pins to pins. This solution can be found by computing the Minimum Spanning Tree (MST) of all possible pin connections. If waypoints, called Steiner points [15], are allowed, the wirelength can be shortened further as illustrated in panel (d).

Due to manufacturing constraints, only rectilinear (horizontal and vertical) wires are typically allowed. Panels (e) through (h) of Fig. 1 show four possible rectilinear solutions. They are constructed by connecting the pins in P2-P1-P3 order using L-shaped wires with one turn. The solutions in panels (e), (f), and (g) all have the same suboptimal wirelength. The solution in panel (h) containing the Steiner point S is optimal. It is shorter because the part of the two L-wires between P1 and S is shared. In general, the goal of Rectilinear-Steiner-Minimum-Tree (RSMT) generation is to interconnect k pins in a plane using only horizontal and vertical wires such that the total wirelength is minimized.

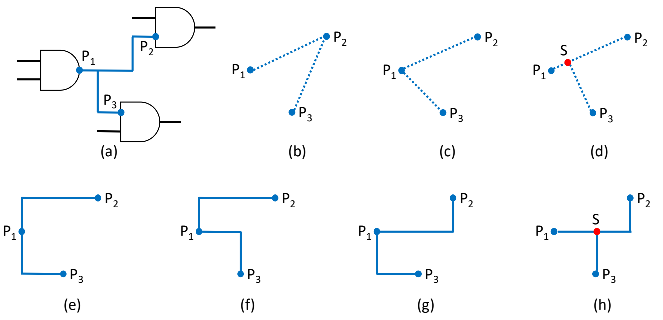

Fig. 1. Several layouts of a three-pin net (blue dots denote pins, red dots indicate Steiner points, the net is blue, and solid wires are rectilinear)

Finding the set of optimal Steiner points is NP-hard [15]. However, interconnecting given pins and Steiner points can be accomplished by computing an MST, a polynomial-time algorithm, of a complete graph where the vertices are the pins and Steiner points, and the edge weights are the rectilinear distance (Manhattan distance) between the edge's endpoints. Despite the NP-hardness, the optimal solution can often be computed in practice because nets tend to comprise just a few pins. Finding the optimal solution is trivial for two-pin nets and easy for three-pin nets (cf. Section III). Taken together, two- and three-pin nets

This material is based upon work supported by the Defense Advanced Research Projects Agency under award number DARPA HR001117S0054. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

often make up the majority of the nets in a netlist. FLUTE [7], a fast and effective RSMT algorithm, precomputes the optimal solution for all nets with up to nine pins, stores them in a table, and performs a lookup when an RSMT is needed. GeoSteiner [22], another RSMT algorithm, computes the optimal solution for any net using linear programming to avoid the exponential runtime in many cases.

We are building a VLSI-design tool chain where every step is parallelized, including the RSMT generation. Many RSMT approaches exist (cf. Section II), but none have been devised with parallel execution in mind. We set out to create a new RSMT heuristic that is deterministic, fast, easy to parallelize, and incurs only a small wirelength penalty. We named the resulting heuristic SFP. It requires little state per thread (O(n) memory for n-pin nets with a small constant factor) and can, therefore, also be used on massively parallel devices like GPUs. SFP is based on a fast approach to compute the optimal solution for three-pin nets, which it applies to larger nets (cf. Section III). On the sixteen ISPD 2008 inputs with between 219,794 and 2,635,625 nets, it incurs an average wirelength penalty of 0.38% relative to FLUTE while running 19.5 times faster on a 16-core system and 86.1 times faster on a GPU.

This paper makes the following contributions.

- It presents SFP, a fast and deterministic RSMT heuristic.
- It describes how to parallelize SFP for CPUs and GPUs.
- It shows that SFP incurs only a small wirelength penalty while running two orders of magnitude faster than FLUTE.

The serial C++, parallel OpenMP, and parallel CUDA implementations of SFP are freely available in open source [20].

The rest of the paper is organized as follows. Section II describes related work. Section III explains our SFP heuristic in detail. Section IV presents the experimental methodology. Section V discusses the performance and wirelength results. Section VI concludes the paper with a summary.

II. Related Work

There is only little prior work on parallel RSMT algorithms. Hence, we also summarize important milestones of serial Steiner-tree generation in the following paragraphs.

The Batched Iterated 1-Steiner (BI1S) heuristic starts by computing all optimal 1-Steiner points [16]. It then finds a maximal independent set of all Steiner points that shorten the wirelength without reducing the MST cost savings of any other points in the set. These Steiner points are incorporated in a single batch. This process repeats until there are no more Steiner points that result in savings. SFP follows the same high-level structure of computing possible Steiner points, eliminating useless and conflicting candidates, and iteratively adding the remaining candidates in batches until convergence is reached. However, SFP uses a new and very different heuristic for identifying the possible Steiner points.

GeoSteiner [21][22] employs a two-phase exact algorithm for creating Full Steiner Trees (FSTs). It first generates a small set of FSTs and then concatenates them using an integer programming formulation solved by branch-and-cut. Zachariasen [23] improves upon this approach by growing the FSTs to avoid overhead in the first phase. Later, he and Rohe [24] propose an algorithm that deletes useless points on the Hanan grid.

Zhou [25] combines the edge substitution idea from Borah et al. [4] with the spanning graph algorithm by Zhou et al. [26] to arrive at an efficient RSMT heuristic that has a worst-case time complexity of O(n log n). As mentioned, FLUTE [7] employs a different approach to reduce runtime. It precomputes optimal solutions for low-degree nets and records them in a lookup table. Additionally, it recursively breaks high-degree nets into smaller subnets until they are small enough for the lookup table. FLUTE provides parameters to control its lookup table size and the accuracy of the recursive step to trade off processing speed for result quality and vice versa.

More recent work extends the GeoSteiner and FLUTE approaches to avoid obstacles. Li et al. [17] modify GeoSteiner to support obstacles in the routing region while maintaining solution optimality. FOARS [2] is an obstacle-aware version of FLUTE that partitions a net into multiple subnets that are uncluttered by obstacles. The solution for each subnet is then merged and refined. Cao et al. [5] introduce Fashion, a solution to the global routing problem that uses dynamic pattern routing and movable-segment-driven dynamic pattern routing. In addition, Fashion also handles overflow. SFP does not handle overflow or obstacles as it focusses on speed and simplicity.

In addition to wirelength, SALT [6] can also minimize signal delay or source-sink pathlength. It is based on a shallow-light tree data structure that enables a smooth tradeoff between the two objectives. Alpert et al. [1] proposes a Prim-Dijkstra-based construction (PD-2), which outperforms SALT in terms of source-sink pathlengths but yields similar wirelengths. We include BI1S, GeoSteiner, FLUTE with different parameters, and SALT in our performance evaluation in Section V.

Most pre-existing RSMT algorithms can be parallelized by dividing up the nets across multiple CPU cores. Yet, we only found two papers that mention parallelism [8][18]. Both of them describe distributed-memory solutions and focus on sorting, partitioning, and communication, three issues that are not of concern in shared-memory implementations such as our work.

Additionally, parallelization for massively parallel devices, like a GPU, is difficult as it requires two levels of parallelization, both over the nets and within the nets. We only found one very old paper that targets a Connection Machine [14]. It assigns each pin to a processor and builds the Steiner trees along scanlines. We are not aware of any prior GPU parallelization of an RSMT algorithm.

III. SFP Heuristic

Fig. 2 depicts optimal solutions for three-pin nets. The solution must lie on the Hanan grid [11], which is outlined in gray. After considering rotational and mirror symmetries as well as the fact that there is no order among the three pins, only the two scenarios shown in the top half of the figure remain (note that the horizontal and vertical distances a, b, c, and d can be zero). Optimal solutions for both scenarios are given in the bottom half of the figure. In either case, the L-routes between pin pairs can be flipped diagonally to obtain other optimal solutions. The scenario shown on the right requires a Steiner point.

Our SFP RSMT heuristic is based on the observation that the center point of any 3-by-3 Hanan grid is always either the only Steiner point or a pin of the net. In both cases, the solution is to directly connect the non-center-point pins to the center point.

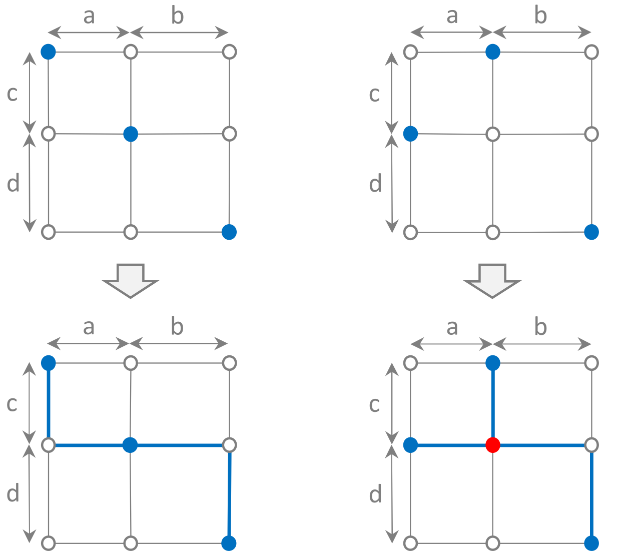

Fig. 2. Possible 3-pin nets (top) and their optimal solutions (bottom) on the Hanan grid; pins and net are blue, the Steiner point is red

Importantly, the coordinates of the center point (x_m, y_m) can be obtained rapidly from the coordinates of the three pins. x_m is the median of the three x coordinates and y_m is the median of the three y coordinates. The median m of three values v_1, v_2, and v_3 can be computed with just four binary operations:

$$m = \max(\min(v_1, v_2), \min(\max(v_1, v_2), v_3))$$

Many CPUs and GPUs have machine instructions to compute the minimum and maximum of two values. They only need to execute four instructions to find the median of three values.

Our SFP heuristic uses this approach directly on three-pin nets to find the optimal solution and *conceptually* applies the idea to larger nets by processing them in the following way:

1. Compute the Minimum Spanning Tree (MST) of all pins based on their pairwise rectilinear distances.
2. Pick any two pins that are connected to the same third pin in the resulting MST.
3. Compute the median of the coordinates of those three pins.
4. If this median location is distinct from the three pins, add a Steiner point (i.e., a new pin) at that location.
5. Repeat the above four steps until no more Steiner points can be added.

Fig. 3 illustrates this procedure on the example of a four-pin net. First, the MST depicted in panel (a) is computed. Then, three connected pins, P_1, P_2, and P_3 in this example, are selected. Their median is S_1 as shown in panel (b). Since S_1 does not coincide with any of the three selected pins, it is added as a Steiner point and a new MST is computed, which is depicted in panel (c). Now, the only three connected pins that yield a distinct median are S_1, P_3, and P_4, so they are selected and Steiner point S_2 is added as shown in panel (d). Finally, a last MST is computed, which is depicted in panel (e). Since all connected sets of three pins now have medians that coincide with one of their pins, the algorithm terminates. Note that the heuristic could alternately have selected P_2, P_3, and P_4 in the first step, which would have led to adding S_2 before S_1.

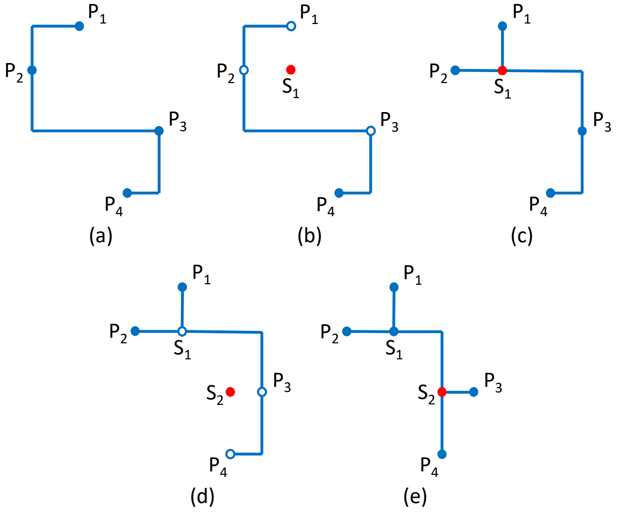

Fig. 3. Conceptual steps of the SFP heuristic applied to a sample 4-pin net (the selected pins have a hollow core)

The SFP heuristic is not guaranteed to yield optimal solutions for nets with more than three pins. The upper bound of the MST solution's suboptimality was proven to be ½ [12]. This implies that any RSMT solution that improves upon the MST solution will share this upper bound. Because SFP is an MST-based improvement heuristic, i.e., it only adds Steiner points when they shorten the wirelength, its upper bound is also at most ½.

Our experimental tests with SFP resulted in the worst case illustrated in Fig. 4, indicating that the upper bound may only be ⅓. In this example, the MST is W-X-Y-Z because the rectilinear distance of 2a+ε between W and Z is greater than the distance of 2a between W and X as well as between Y and Z. Given this MST, the SFP heuristic cannot find a Steiner point that is not already a pin of the net. However, the optimal solution does require a Steiner point as shown in the right panel.

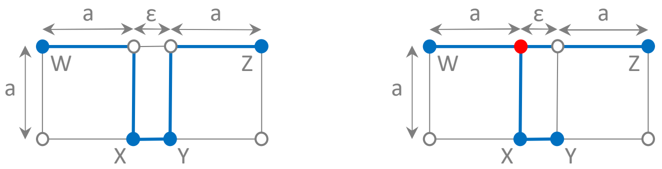

Fig. 4. Four-pin example with $\varepsilon < a$: MST-based solution on left, optimal solution on right

The wirelength of SFP's solution is 4a+ε whereas the optimal wirelength is 3a+2ε. The limit of the wirelength ratio is:

$$\lim_{\varepsilon\to 0}\frac{4a+\varepsilon}{3a+2\varepsilon}=\frac{4a}{3a}=\frac{4}{3}$$

This experimental bound of ⅓ is consistent with the results presented in Section V.B below.

The scenario in Fig. 4 occurs almost never in practice. We added a fix for it to our code but found it to lower the overall wirelength only insignificantly on the 16 tested inputs. We removed the fix so as not to slow down and complicate the implementation. It is not included in the rest of the paper.

Algorithm 1 SFP heuristic

Input: *N,* the pin net to be processed
Output: *G*, the processed pin net with the edges and the Steiner points added (i.e., the RSMT)

```
1:  procedure SFP(N)
2:    repeat
3:      pinAdded ← False
4:      G ← MST(N, rectilinear)
5:      for each edge e in G do
6:        dist[e] ← 0
7:        SteinerPoint[e] ← ∅
8:      end for
9:      for each edge pair ($e_1$, $e_2$) where $e_1$ connects
              $p_1$, $p_2$ and $e_2$ connects $p_2$, $p_3$ in G do
10:       $p_s$ ← MedianCoordinates($p_1$, $p_2$, $p_3$)
11:       if $p_s \neq p_1$ and $p_s \neq p_2$ and $p_s \neq p_3$ then
12:         if Dist($p_2$, $p_s$) > dist[$e_1$] then
13:           dist[$e_1$] ← Dist($p_2$, $p_s$)
14:           SteinerPoint[$e_1$] ← $p_s$
15:         end if
16:         if Dist($p_2$, $p_s$) > dist[$e_2$] then
17:           dist[$e_2$] ← Dist($p_2$, $p_s$)
18:           SteinerPoint[$e_2$] ← $p_s$
19:         end if
20:       end if
21:     end for
22:     for each edge pair ($e_1$, $e_2$) where $e_1$ connects
              $p_1$, $p_2$ and $e_2$ connects $p_2$, $p_3$ in G do
23:       if dist[e1] > 0 then
24:         if SteinerPoint[$e_1$] == SteinerPoint[$e_2$] then
25:           N.include(SteinerPoint[$e_1$])
26:           pinAdded ← True
27:         end if
28:       end if
29:     end for
30:   until pinAdded == False
31:   return G
32: end procedure
```

A. Serial Implementation

The full SFP algorithm is given in Algorithm 1. For performance reasons, it differs a little from the conceptual approach outlined earlier. Since finding all connected pin triples and computing their medians can be done in linear time, but generating the MST requires quadratic time, our implementation minimizes the number of times an MST is computed (line 4). Specifically, it always computes all potential Steiner points in each round (repeat-until loop, lines 2 through 30) and batch adds (lines 22 through 29) those that reduce the wirelength (line 23) while not conflicting with "better" Steiner points (line 24). To find these "best" Steiner points, SFP determines the median coordinates (line 10) for each pin triple (line 9). If the median is distinct (line 11) and provides more wirelength savings (lines 12 and 16), the potential Steiner point is recorded on the two edges (lines 14 and 18) along with its savings (lines 13 and 17).

Fig. 5 illustrates the operation of the SFP heuristic step-by-step on a six-pin net. Panel (a) displays the MST of the six given pin locations. Each diagonal (dotted) line in the figure should be thought of as a rectilinear L-route. Panel (b) shows, using crosshairs, the four potential Steiner-point locations computed by SFP when considering the four connected pin triples DEF, CDE, BCD, and ABC. Potential Steiner points that involve a common edge (each pin triple encompasses two edges) are in conflict and will not be included together (see below). In case of conflict, SFP picks the Steiner point that is farther away from the median point of the triple because this maximizes the wirelength savings. Ties are broken deterministically using the unique pin IDs. As seen in panel (b), only Steiner point S3 is selected. It conflicts with the potential Steiner points S2 and S4, which provide less savings as S2 is closer to pin D and S4 is closer to pin B than S3 is to pin C. Moreover, the potential Steiner point S1 conflicts with the potential Steiner point S2 but provides less wirelength savings as S1 is closer to pin E than S2 is to pin D, so S1 is also deselected.

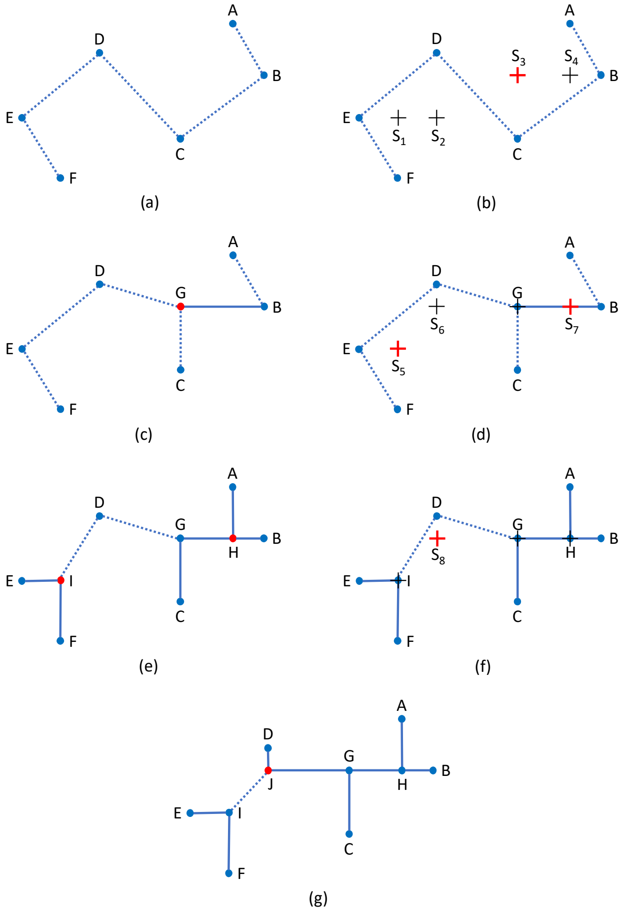

Fig. 5. Step-by-step processing of a six-pin net using the SFP heuristic

After adding the new Steiner point (S3 → G), the MST in panel (c) is computed. Panel (d) shows the next set of potential Steiner points. Some of them coincide with G and are, therefore, not considered. However, two of the remaining three potential Steiner points are independent and provide the locally maximal wirelength savings, so both S5 and S7 are included at the same time, that is, they are added in a single batch.

The MST with the newly added pins H and I included is given in panel (e). As shown in panel (f), all but one of the potential Steiner points coincide with existing pins, so only the one Steiner point (S8 → J) is considered and included. The final MST is presented in panel (g). Since all potential Steiner points now coincide with existing pins, the SFP heuristic terminates.

Fig. 6 provides an example of four medians (potential Steiner points) whose corresponding pin triples each share a common edge with two other pin triples. Only two opposing medians should be used as Steiner points because including all four medians would yield the suboptimal solution shown on the right. This example highlights why SFP checks for conflicts and uses at most one median per MST edge at a time.

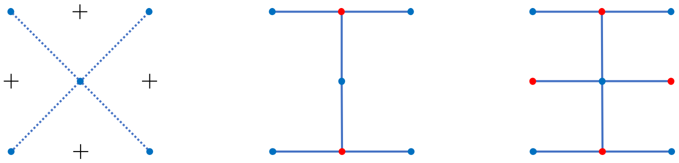

Fig. 6. Five-pin net with MST and 4 medians (left), optimal solution with 2 Steiner points (center), and poor solution using all 4 medians (right)

Our SFP implementation employs Prim's algorithm to compute the MST. For a net with n pins, this requires $O(n^2)$ work since the graph is complete. Determining all connected pin triples and computing their medians, i.e., finding the potential Steiner points, requires O(n) work as the work per tree edge is constant. Hence, each iteration of SFP requires $O(n^2)$ work. In the best case, O(1) iterations suffice. In the worst case, O(n) iterations are needed because there are at most O(n) Steiner points and at least one new Steiner point must be added per iteration. This leads to a worst-case running time of $O(n^3)$. The empirical test with large random nets described in Section V.D yielded an average-case running time of about $O(n^{2.17})$ because, in most iterations, many Steiner points are added at the same time. Here, it is important to note that big-O complexity analysis only holds for sufficiently large n. In the RSMT domain, almost every net has a very small n, meaning that other factors dominate and determine the runtime, making the big-O complexity largely irrelevant.

B. *Parallel Implementations*

Our parallel SFP codes implement the same algorithm as the serial code. In fact, the parallel codes are deterministic and always produce the exact same result as the serial SFP code.

1) OpenMP CPU Parallelization

Since a netlist typically comprises many nets but each net tends to only have a few pins, we opted to parallelize SFP across nets in our OpenMP implementation for multicore CPUs. In other words, each thread applies the heuristic to a separate set of nets by running Algorithm 1 on each net. Since nets require different amounts of processing time, we use a dynamic schedule to balance the workload. As soon as a thread is done with one net, it obtains the next net from the worklist. This procedure repeats until all nets have been processed.

2) CUDA GPU Parallelization

This straightforward parallelization does not work well for GPUs. We tried it and found it to underperform the OpenMP code due to thread divergence, little opportunity for coalesced memory accesses, too large of an active working set, and too little shared memory space for each simultaneously processed net. To alleviate these problems and boost performance, our CUDA implementation assigns each net to an entire warp, again using a dynamic schedule for load balancing. The 32 threads making up a warp process the given net in parallel, i.e., both the MST and the Steiner-point computations are parallelized as explained in the next two paragraphs. Hence, the GPU code combines two levels of parallelization: (1) within each net and (2) across all nets. Nevertheless, the kernel code that executes on the GPU only requires about 100 CUDA statements.

We parallelized Prim's MST algorithm (called on line 4 in Algorithm 1) as follows. The code comprises a doubly nested loop that performs $O(n^2)$ work. Due to a loop-carried dependency in the outer loop, we targeted the inner loop. This loop updates the distance from the growing tree to all pins that are not yet in the tree and determines the closest such pin. The distance computations are independent and easy to parallelize. Finding the pin with the minimum distance requires a reduction, which is implemented using atomicMin operations.

The Steiner-point computation is parallelized as follows. The initialization is embarrassingly parallel (loop starting at line 5 in Algorithm 1). To simplify the rest of the computation, the code first converts the edge list into an adjacency list for each pin, i.e., it builds a more suitable data structure. Each warp thread processes a different set of edges and appends the two neighboring pins to the appropriate adjacency lists. AtomicAdd operations are used to determine the correct location in the adjacency lists. Then, the code computes the median of the three pins for all edge pairs with a common pin (loop starting at line 9). Each warp thread processes a disjoint set of adjacency lists. The highest achievable distance savings is recorded for each edge using an atomicMax operation (if statements starting at lines 12 and 16). Finally, the warp threads concurrently process the edge pairs to determine which pairs agree on the same Steiner point (loop starting at line 22). Those Steiner points are added to the list of pins using a parallel sum reduction to determine their location in the list.

The resulting C++, OpenMP, and CUDA codes are available in open source [20]. All three versions, including the parallel ones, produce the same output for the same input and are deterministic, meaning that they always produce the same result for a given input irrespective of what system they are executed on or what thread count is being used. Moreover, SFP naturally handles duplicate pins in the input, so there is no need to remove duplicates before processing.

IV. Experimental Methodology

We evaluated the seven RSMT codes shown in TABLE I. We selected these codes because they perform well and are publicly available. FLUTE is fast and produces an optimal solution for 9-pin and smaller nets. GeoSteiner is slow but guaranteed to produce an optimal solution for all nets. BI1S lies between GeoSteiner and FLUTE in terms of both speed and result quality. Similarly, SALT lies between FLUTE and BI1S.

TABLE I EVALUATED RSMT IMPLEMENTATIONS

Device	Ser/Par	Name	Version	Origin
CPU	serial	BI1S	n/a	[3]
		FLUTE	3.1	[9]
		GeoSteiner	5.1	[10]
		SALT	n/a	[19]
		SFP-C++ (our code)	1.0	[20]
CPU	parallel	SFP-OpenMP (our code)	1.0	[20]
GPU	parallel	SFP-CUDA (our code)	1.0	[20]

We modified the programs so that they all use the same code to read the inputs into memory, time the RSMT computation, and evaluate the wirelength. In other words, the programs only differ in their implementation of the RSMT algorithm. Note that only the RSMT computation is timed. For FLUTE, the time to load the lookup table is not included. To make SALT compatible with the other codes, we set ε=10,000 so it minimizes the wirelength. We ran each program five times and used the median of the measured times to compute the throughput. Since all seven implementations are deterministic, they always produce the same wirelength for a given input. We checked the output by comparing the solution for each net to that of GeoSteiner. As expected, none of the other implementations produced a better solution for any net.

The system we used for the measurements is based on an AMD Ryzen Threadripper 2950X CPU with 16 cores. Hyperthreading is enabled, i.e., the cores can simultaneously run 32 threads. The main memory has a capacity of 64 GB. The operating system is Fedora 30. The GPU in the system is an NVIDIA Titan V with 5120 processing elements distributed over 80 multiprocessors. Its global memory has a capacity of 12 GB. The GPU driver version is 450.66.

We compiled the CPU codes with gcc/g++ 9.3.1 using the "-O3 -march=native" optimization flags and the GPU code with nvcc 11.0 using the "-O3 -arch=sm_35" optimization flags. The OpenMP version of our code uses 32 threads and employs a dynamic schedule (for load balancing) with a chunk size of 256 (to minimize overhead and false sharing). The CUDA version also employs a dynamic schedule. It processes nets with up to 32 pins using 61,440 threads and the rest with 7,680 threads (to increase the amount of shared memory available per thread).

We used the sixteen inputs listed in TABLE II. They stem from the ISPD 2008 Global Routing Contest [13]. The table lists the name, abbreviation, total number of nets, how many nets have more than 256 pins, the average net size, and the maximum net size for each input. For all codes, we truncated the few nets with more than 256 pins and only used the first 256 pins. We had to truncate because, as downloaded, BI1S and GeoSteiner only work correctly for input nets up to and including 256 pins but not for all nets up to 512 pins. This affected between two and six nets per input. Moreover, we had to remove duplicate pins from the inputs as SALT does not support duplicates.

TABLE II INPUT NETLIST INFORMATION

Input	Abbrev.	Number of nets	Nets with >256 pins	Average pins per	Pins in largest
adaptec1	a1	219,794	2	4.28	2,271
adaptec2	a2	260,159	2	4.08	1,935
adaptec3	a3	466,295	2	4.01	3,713
adaptec4	a4	515,304	2	3.70	3,974
adaptec5	a5	867,441	4	4.01	9,863
bigblue1	b1	282,974	2	4.03	2,621
bigblue2	b2	576,816	3	3.65	11,869
bigblue3	b3	1,122,340	2	3.40	7,623
bigblue4	b4	2,228,903	4	3.98	20,766
newblue1	n1	331,663	2	3.68	12,335
newblue2	n2	463,213	2	3.80	8,089
newblue3	n3	551,667	2	3.42	16,008
newblue4	n4	636,195	2	3.90	13,518
newblue5	n5	1,257,555	6	3.90	17,912
newblue6	n6	1,286,452	2	4.10	19,862
newblue7	n7	2,635,625	2	3.82	17,324

V. RESULTS

A. Runtime and Throughput

TABLE III lists the time it takes the seven programs on our system to process all nets in each input. The bottom row shows the geometric mean over the sixteen inputs. The corresponding throughput, i.e., the number of nets divided by the processing time, is listed in TABLE IV. Higher throughputs are better. The rightmost three columns present results for variations of FLUTE. For example, "8-1" corresponds to a lookup table size of 8 and an accuracy parameter of 1. The default FLUTE lookup table size is 9 with an accuracy of 3. We do not show FLUTE 9-2 as it performs between 9-1 and 9-3 in both wirelength and speed and, therefore, does not provide additional insight.

Compared to the serial SFP$_{\text{C++}}$ version, SFP$_{\text{OMP}}$ running on 16 hyperthreaded cores is 16 times faster (i.e., hyperthreading does not help much) and SFP$_{\text{CUDA}}$ running on the Titan V GPU is 71 times faster, showing that our algorithm is parallelizable and scales to many cores and processing elements.

On each tested input, our parallel GPU code is the fastest and our parallel CPU code is the second fastest. Ignoring the non-default versions of FLUTE, our serial CPU code is the third fastest implementation with one exception. FLUTE is the next, followed by SALT, then BI1S, and GeoSteiner is the slowest. Our serial SFPC++ code is 1.21 times faster than default FLUTE but only a little more than half as fast as FLUTE with a lower accuracy (9-1). Interestingly, reducing the table size of FLUTE (8-1) does not yield additional speedup. In contrast, increasing the accuracy to the maximum level (9-12) makes FLUTE nearly 15 times slower. Evidently, the accuracy parameter has a significant impact on the throughput of FLUTE whereas the size of the lookup table does not. SFP$_{\text{OMP}}$ is 19.5 times faster and SFP$_{\text{CUDA}}$ is 86.1 times faster than the serial default FLUTE.

TABLE III RUNTIME (MILLISECONDS)

	BI1S	FLUTE	GeoSt.	SALT	SFPC++	SFPOMP	SFPCUDA	8-1	9-1	9-12
a1	6,127.9	174.7	15,385.3	3,047.5	149.4	11.2	3.0	73.1	73.0	1,881.4
a2	8,521.1	199.8	18,898.9	3,026.7	157.2	10.6	4.0	73.7	73.5	2,638.6
a3	8,742.8	210.7	20,714.9	3,433.9	175.5	10.6	2.0	83.1	83.4	2,928.3
a4	13,353.2	357.9	29,988.7	5,473.5	259.1	16.2	3.8	128.2	128.1	4,644.1
a5	21,025.1	465.9	42,065.6	6,884.6	357.8	20.2	3.9	167.5	166.8	6,065.7
b1	21,042.1	527.7	55,448.8	10,986.8	530.7	36.7	7.6	261.3	261.3	6,838.0
b2	12,307.5	194.4	20,785.5	3,208.5	163.4	9.8	2.0	80.1	79.8	2,945.3
b3	12,574.5	302.0	29,647.2	5,813.3	263.6	15.2	2.7	133.0	132.9	4,608.2
b4	64,901.4	1,139.0	108,943.9	15,341.9	882.2	45.9	9.1	375.3	374.7	16,094.1
n1	51,845.8	1,353.4	123,745.4	24,588.6	1,125.0	93.9	29.2	578.8	577.6	14,528.2
n2	9,658.5	251.7	22,433.9	4,011.0	193.8	12.4	3.2	94.6	94.2	3,228.8
n3	15,006.3	294.5	24,123.2	4,729.6	218.6	15.9	2.6	110.7	110.3	4,188.9
n4	16,779.2	379.2	34,741.9	6,564.7	312.3	19.0	4.2	156.1	157.1	4,911.6
n5	25,996.9	507.6	51,799.0	8,170.1	441.6	24.4	5.3	203.3	202.8	7,218.6
n6	46,089.6	1,094.5	104,726.4	17,039.6	887.3	47.4	10.5	409.8	408.4	15,242.5
n7	5,464.4	133.2	13,085.3	2,566.4	123.6	6.6	1.8	360.4	358.7	11,039.9
gm	16,193.6	369.7	34,820.9	6,142.3	304.8	19.0	4.3	164.4	164.2	5,507.7

TABLE IV THROUGHPUT (MILLIONS OF NETS PER SECOND)

	BI1S	FLUTE	GeoSt.	SALT	SFPC++	SFPOMP	SFPCUDA	8-1	9-1	9-12
a1	0.04	1.26	0.01	0.07	1.47	19.62	73.86	3.01	3.01	0.12
a2	0.03	1.10	0.01	0.07	1.40	20.77	55.13	2.98	2.99	0.08
a3	0.03	1.23	0.01	0.08	1.48	24.59	129.78	3.13	3.12	0.09
a4	0.03	1.30	0.02	0.09	1.80	28.71	123.54	3.64	3.64	0.10
a5	0.02	1.11	0.01	0.07	1.44	25.48	131.69	3.08	3.09	0.08
b1	0.04	1.64	0.02	0.08	1.63	23.65	114.10	3.32	3.32	0.13
b2	0.02	1.46	0.01	0.09	1.73	28.85	141.62	3.53	3.55	0.10
b3	0.05	1.91	0.02	0.10	2.19	37.97	215.26	4.34	4.34	0.13
b4	0.02	0.99	0.01	0.07	1.27	24.44	123.23	2.99	3.00	0.07
n1	0.04	1.65	0.02	0.09	1.98	23.73	76.26	3.85	3.86	0.15
n2	0.03	1.32	0.01	0.08	1.71	26.64	103.23	3.50	3.52	0.10
n3	0.03	1.57	0.02	0.10	2.12	29.09	178.13	4.19	4.20	0.11
n4	0.03	1.45	0.02	0.08	1.77	28.99	131.99	3.53	3.51	0.11
n5	0.02	1.25	0.01	0.08	1.44	26.07	119.73	3.13	3.14	0.09
n6	0.03	1.15	0.01	0.07	1.42	26.51	120.24	3.07	3.08	0.08
n7	0.04	1.65	0.02	0.09	1.78	33.41	122.30	3.57	3.59	0.12
gm	0.03	1.36	0.01	0.08	1.64	26.44	116.81	3.41	3.41	0.10

In absolute terms, our GPU implementation processes between 55 million and 215 million nets per second on the ISPD 2008 inputs. It provides nearly two orders of magnitude in speedup over FLUTE, three orders over SALT, well over three orders over BI1S, and close to four orders over GeoSteiner.

BI1S, FLUTE, GeoSteiner, and SALT are likely parallelizable. Whereas truly parallelizing them is beyond the scope of our project, we did run 32 copies of each serial code, including the SFP code, simultaneously and divided the resulting median runtime by 32 to approximate the parallel performance. This approach disregards any synchronization that might be necessary in an actual parallelization. The corresponding throughputs for each input are presented in TABLE V.

This pseudo-parallelization yields a speedup of about 17 for SALT, 19 for FLUTE, and 20 for BI1S, GeoSteiner, and SFP. Even under these optimistic conditions, FLUTE is still slower than the OpenMP version of SFP (SFP_{OMP}) on the majority of the inputs and on average when running on the CPU. Our GPU implementation is 4.6 times faster than pseudo-parallel FLUTE. Note that SFP-32 is 27.5% faster than SPF_{OMP}, providing an indication of how optimistic the pseudo-parallel throughputs are. In other words, the throughputs of genuine parallel implementations of BI1S, FLUTE, GeoSteiner, and SALT would probably be significantly lower than the reported pseudo-parallel throughputs.

TABLE V THROUGHPUT (MILLIONS OF NETS PER SECOND) WHEN RUNNING 32 COPIES OF EACH SERIAL CODE SIMULTANEOUSLY COMPARED TO PARALLEL SFP

	BI1S-32	FLUTE-32	GeoSt-32	SALT-32	SFP-32	SFPOMP	SFPCUDA
a1	0.71	23.62	0.29	1.38	30.69	19.62	73.86
a2	0.51	20.84	0.23	1.37	29.06	20.77	55.13
a3	0.58	23.33	0.25	1.45	30.35	24.59	129.78
a4	0.68	24.72	0.31	1.63	37.12	28.71	123.54
a5	0.48	21.07	0.25	1.44	29.26	25.48	131.69
b1	0.82	30.15	0.32	1.50	34.69	23.65	114.10
b2	0.45	27.23	0.27	1.68	34.87	28.85	141.62
b3	0.90	34.68	0.39	1.89	44.69	37.97	215.26
b4	0.33	18.17	0.21	0.78	25.53	24.44	123.23
n1	0.85	29.48	0.36	1.72	40.18	23.73	76.26
n2	0.68	25.16	0.30	1.55	35.03	26.64	103.23
n3	0.60	30.18	0.39	1.85	42.35	29.09	178.13
n4	0.65	27.33	0.32	1.59	36.74	28.99	131.99
n5	0.48	23.87	0.25	1.49	29.67	26.07	119.73
n6	0.53	21.97	0.24	1.42	29.13	26.51	120.24
n7	0.78	30.05	0.34	0.55	36.31	33.41	122.30
gm	0.61	25.39	0.29	1.40	33.72	26.44	116.81

B. Wirelength

The total wirelength, i.e., the sum of the wirelength over all nets, is listed in TABLE VI. Higher numbers are worse. Recall that each netlist has between two and six nets that were truncated. The results for GeoSteiner are absolute, the remaining results are relative to GeoSteiner. Only one value is listed for SFP as our three implementations always produce the same result for a given input. The bottom row shows the geometric mean over all inputs. As before, we include the results of FLUTE with different parameters in the three right-most columns.

TABLE VI TOTAL WIRELENGTH (ABSOLUTE FOR GEOSTEINER, RELATIVE INCREASE OVER GEOSTEINER FOR THE OTHER ALGORITHMS, I.E., WIRELENGTH = (1 + PERCENTAGE) * GEOSTEINER)

	GeoSteiner	BI1S / GS	FLUTE / GS	SALT / GS	SFP / GS	8-1 / GS	9-1 / GS	9-12 / GS
a1	118,774,718	0.11%	0.20%	0.12%	0.74%	0.60%	0.55%	0.06%
a2	112,533,313	0.08%	0.18%	0.10%	0.58%	0.56%	0.52%	0.04%
a3	280,751,697	0.06%	0.18%	0.09%	0.53%	0.48%	0.45%	0.06%
a4	266,751,075	0.06%	0.19%	0.09%	0.48%	0.57%	0.54%	0.06%
a5	491,276,114	0.07%	0.19%	0.09%	0.58%	0.60%	0.56%	0.05%
b1	171,900,004	0.10%	0.27%	0.14%	0.83%	0.74%	0.68%	0.05%
b2	184,719,520	0.11%	0.25%	0.14%	0.60%	0.69%	0.65%	0.06%
b3	386,392,738	0.06%	0.17%	0.08%	0.46%	0.42%	0.39%	0.05%
b4	939,957,856	0.09%	0.20%	0.11%	0.60%	0.64%	0.60%	0.06%
n1	70,066,240	0.10%	0.16%	0.09%	0.60%	0.53%	0.49%	0.04%
n2	231,008,765	0.07%	0.15%	0.08%	0.47%	0.47%	0.44%	0.04%
n3	295,740,999	0.04%	0.10%	0.05%	0.28%	0.32%	0.30%	0.03%
n4	319,102,815	0.06%	0.11%	0.06%	0.49%	0.36%	0.33%	0.03%
n5	569,689,521	0.10%	0.28%	0.15%	0.69%	0.77%	0.72%	0.06%
n6	587,289,719	0.08%	0.19%	0.10%	0.56%	0.56%	0.52%	0.05%
n7	1,429,968,074	0.07%	0.15%	0.08%	0.53%	0.45%	0.41%	0.04%
gm	299,290,278	0.08%	0.18%	0.10%	0.55%	0.53%	0.50%	0.05%

Compared to GeoSteiner, which computes the optimal solution, the other approaches are worse on every input. Disregarding the non-default versions of FLUTE, BI1S is second best, SALT is third, FLUTE is fourth, and SFP is last. However, the differences are small. The total wirelength of BI1S is only 0.08% longer than GeoSteiner, SALT's is 0.1% longer, FLUTE's is 0.18% longer, and SFP's is 0.55% longer. In other words, SFP's wirelength is about half a percent longer than optimal on average and less than one percent longer on all tested inputs. We believe this to be a small increase for a solution that is computed over 8100 times faster on average (cf. TABLE IV). Moreover, given the simplicity of the SFP heuristic, the wirelength is surprisingly good. Compared to FLUTE, the fastest of the other four tested programs, SFP's wirelength is 0.38% longer (and computed 86 times faster). This small amount of extra wirelength is likely insignificant compared to the wirelength added in the later routing stage, in particular during rip-up and re-route.

The results for the variations of FLUTE show that the parameters, especially the accuracy parameter, allow the user to trade off throughput (cf. TABLE IV) for wirelength. Both accuracy-1 versions of FLUTE produce a solution that is about 0.5% worse than optimal and only slightly better than SFP. The accuracy-12 version produces a wirelength that is very close to optimal and better than BI1S and SALT while still being faster.

A net-to-net comparison over all inputs (13.7 million nets in total) shows that, in the most extreme case, BI1S, FLUTE, and GeoSteiner find a solution that is 31.1% shorter than SFP's (cf. Fig. 4 and its discussion). In the other extreme, SFP finds a solution that is 10.2% shorter than SALT's, 15.6% shorter than BI1S', and 19.1% shorter than FLUTE's. These improvements again highlight the good result quality that the simple SFP heuristic can deliver. SFP always produces the optimal solution on 2- and 3-pin nets, FLUTE on 2- through 9-pin nets, SALT on 2- through 9-pin nets as it is based on FLUTE, and BI1S on 2-, 3-, and 4-pin nets. GeoSteiner is always optimal.

C. *Wirelength and Throughput for Different Net-Size Ranges*

In this section, we study the wirelength and throughput for small, medium, and large nets separately. We classify nets with 2 or 3 pins as small, nets with 4 to 9 pins as medium, and nets with 10 to 256 pins as large. We selected these ranges such that SFP is optimal on the small nets, FLUTE and SALT on the small and medium nets, and GeoSteiner on all three size ranges.

The total wirelength, that is, the sum of the final wirelength of each net in each size range, is listed in TABLE VII. Higher numbers are worse. To improve readability, we only show the geometric mean over the 16 inputs. The wirelengths for GeoSteiner are absolute whereas the other results are relative to GeoSteiner. Only one value is listed for SFP as our three codes are deterministic and produce the same result.

TABLE VII TOTAL WIRELENGTH (ABSOLUTE FOR GEOSTEINER, RELATIVE INCREASE OVER GEOSTEINER FOR THE OTHER ALGORITHMS) FOR DIFFERENT NET-SIZE RANGES

Pin Range	GeoSteiner	BI1S / GS	FLUTE / GS	SALT / GS	SFP / GS	8-1 / GS	9-1 / GS	9-12 / GS
2 to 3	122,205,140	optimal	optimal	optimal	optimal	optimal	optimal	optimal
4 to 9	81,944,821	0.05%	optimal	optimal	0.51%	optimal	optimal	optimal
10 to 256	89,649,128	0.21%	0.60%	0.32%	1.37%	1.67%	1.67%	0.16%

The table shows that all tested approaches find the optimal solutions on the small nets, only FLUTE, SALT, and GeoSteiner are optimal on the medium nets, and only GeoSteiner is optimal on the large nets as expected. BI1S is off by 0.05% and SFP is off by 0.51% on the medium nets. On the large nets, BI1S is off by 0.21% and outperforms FLUTE, which is off by 0.6%, as well as SALT, which is off by 0.32%. SFP is off by 1.37%. Interestingly, the faster versions of FLUTE with an accuracy of 1 are off by 1.67% on the large nets and thus perform worse than SFP. Except for GeoSteiner, the tested approaches perform progressively worse for larger nets. Nonetheless, they all produce solutions that are close to the optimal wirelength.

TABLE VIII shows the runtimes and TABLE IX the throughputs on the different net-size ranges. Again, we only list the geometric mean over the 16 inputs. The first five (and the last three) codes run on a single CPU core, SFP_{OMP} runs on 16 hyper-threaded CPU cores, and SFP_{CUDA} runs on the GPU. The SFP runtimes vary by no more than a few percent between runs.

TABLE VIII RUNTIME (MILLISECONDS) FOR DIFFERENT NET-SIZE RANGES

Pin Range	BI1S	FLUTE	GeoSt.	SALT	$\text{SFP}_{\text{C++}}$	SFP_{OMP}	SFP_{CUDA}	8-1	9-1	9-12
2 to 3	133.4	37.6	1,915.4	1,868.3	12.0	6.7	0.9	FLUTE	FLUTE	FLUTE
4 to 9	4,413.2	190.8	16,221.5	1,662.0	379.8	35.8	4.9	190.1	FLUTE	FLUTE
10 to 256	353,723.3	7,464.6	743,702.2	4,363.0	5,704.8	412.1	83.3	2,222.8	2,224.0	118,167.0

TABLE IX THROUGHPUT (MILLIONS OF NETS PER SECOND) FOR DIFFERENT NET SIZES

Pin Range	BI1S	FLUTE	GeoSt.	SALT	$\text{SFP}_{\text{C++}}$	SFP_{OMP}	SFP_{CUDA}	8-1	9-1	9-12
2 to 3	4.903	17.394	0.341	0.350	54.287	97.196	715.585	FLUTE	FLUTE	FLUTE
4 to 9	0.148	3.427	0.040	0.393	1.722	18.254	134.270	3.440	FLUTE	FLUTE
10 to 256	0.002	0.088	0.001	0.150	0.115	1.587	7.855	0.294	0.294	0.006

GeoSteiner has the lowest and BI1S the second lowest throughput except on the small nets, where BI1S outperforms SALT. SALT is next followed by FLUTE, which beats serial SFP on medium-sized nets, i.e., the nets where SFP is not optimal, but FLUTE is. Parallel SFP is the second fastest and SFP running on the GPU delivers the highest throughput on each net-size range. SFP_{OMP} does not scale well on the small nets due to the very fast runtime of serial $\text{SFP}_{\text{C++}}$ (just 0.012 seconds for the entire netlist on average), which makes it hard to overcome the parallelization overhead.

Serial SFP ($\text{SFP}_{\text{C++}}$) is over three times faster than FLUTE on the small nets, illustrating how fast the simple median computation is upon which SFP is based. Since it yields the optimal solution, this approach should generally be used for 3-pin nets. However, serial SFP is only half as fast as FLUTE on the medium nets. In this size range, FLUTE benefits from its precomputed solutions, i.e., the table lookup is about twice as fast as computing the solution from scratch as SFP does. Since FLUTE only has precomputed solutions for up to 9-pin nets, this advantage is lost on the large nets, where serial SFP is again faster. However, SALT and especially the accuracy-1 versions of FLUTE are faster than serial SFP on the large nets.

Parallel SFP outperforms the default version of the other four codes on the tested net sizes by at least a factor 5.3 (OpenMP) and 39.2 (CUDA). The two parallel SFP versions exhibit the lowest drop in throughput between the small and the large nets, meaning they scale better to larger nets. This is likely due to the parallelization overhead being lower, relatively speaking, as the absolute overhead per net is roughly constant regardless of the net size but the computation time is higher for larger nets.

D. SFP Performance on Huge Nets

In this section, we evaluate SFP on nets with between 1000 and 10,000 pins. As downloaded, BI1S, FLUTE, GeoSteiner, and SALT do not support such net sizes. For each size, we generated 100 nets with random *x* and *y* coordinates as pin locations in the range $0 \leq x < 1000$ and $0 \leq y < 1000$.

Fig. 7 shows the average absolute runtime per net of serial SFP. We also include the best-fit trendline in the figure to estimate the growth rate.

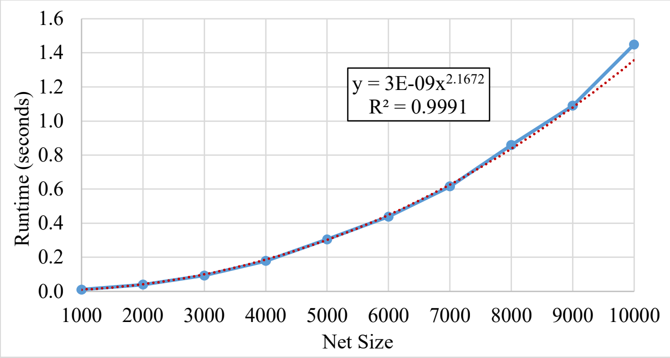

Fig. 7. Average runtime per net of serial SFP on nets with between 1000 and 10,000 randomly placed pins, including the best-fit trendline

These results highlight two important aspects. First, it takes SFP under 1.5 seconds to produce an RSMT for a 10,000-pin net. Hence, SFP can also be used for very large nets that are not supported by the other studied codes. Second, in this net-size range, SFP's runtime complexity appears to be about O($n^{2.17}$), where *n* is the number of pins, based on the best-fit power trendline. This is well below the O(n^3) worst case mentioned in Section III, implying that SFP adds a large batch of Steiner points in each iteration.

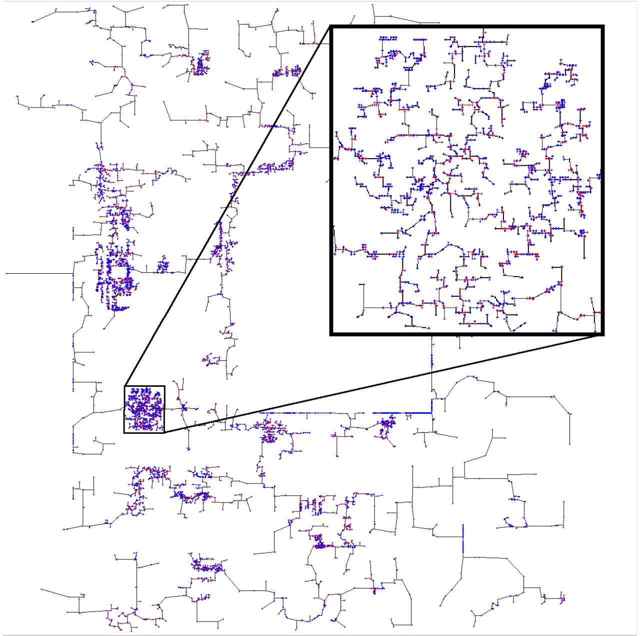

Fig. 8. The RSMT generated by SFP for the largest net in adaptec5; given pins are blue and inserted Steiner points are red (diagonal wires can be replaced with either of the two possible L routes to produce a rectilinear solution)

To show a real-world example of a large net, we ran SFP on the largest net in the adaptec5 input, which has 9,863 pins. Fig. 8 presents the result with one of the dense areas blown up to provide more detail. It took serial SFP just 1.1 seconds and seven iterations to compute this solution, which is 6.5% shorter than the pure MST solution.

Fig. 9 displays the average number of iterations needed by SFP per net to produce the RSMT. We again include the best-fit trendline. The results show that the number of iterations is very small. None of the 100 tested random 10,000-pin nets require more than 14 iterations. Furthermore, the number of iterations grows very slowly with the net size. Taken together, the results presented in this section indicate that SFP can likely be used for computing RSMTs of even larger nets in just seconds.

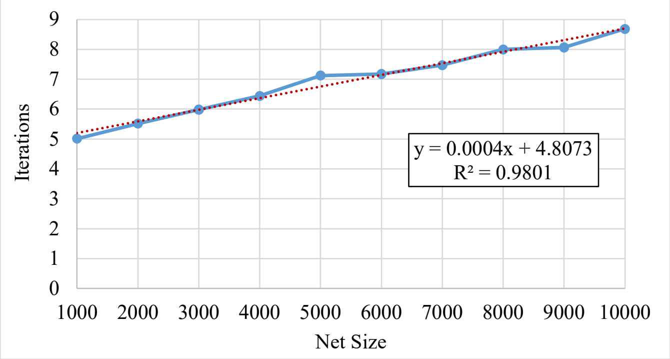

Fig. 9. Average number of iterations per net of SFP on nets with 1000 to 10,000 randomly placed pins, including the best-fit trendline

VI. Summary and Conclusions

We are working toward a VLSI-design tool chain that is fully parallelized. This paper presents one component thereof, a deterministic Rectilinear-Steiner-Minimum-Tree heuristic called SFP and describes how we designed it to be effective and parallelism friendly. It is based on a simple and fast approach to compute the optimal solution for three-pin nets, which it extends to larger nets. Compared to FLUTE, a serial code, SFP incurs a wirelength penalty of 0.38% on the sixteen ISPD 2008 netlists but is 86 times faster when running on a GPU. Compared to GeoSteiner, BI1S, and SALT, SFP's wirelength penalty is no more than 0.55%, but it is up to 8100 times faster. Despite its simplicity, the SFP heuristic can produce solutions that are over 19% shorter than FLUTE's. Moreover, SFP supports much larger net sizes than the other tested codes. The serial C++, parallel OpenMP, and parallel CUDA implementations of SFP are publicly available [20].

If optimal solutions are required, SFP should be used for small nets with 2 or 3 pins, FLUTE for medium nets with 4 to 9 pins, and GeoSteiner for large nets with 10 to 256 pins as they are faster than the other tested codes. In cases where speed is important in addition to a good wirelength and only serial execution is possible, SFP should be used for small inputs, FLUTE for medium inputs, FLUTE with reduced accuracy for large inputs, and SFP for inputs with more than 256 pins. If parallel execution, in particular GPU acceleration, is available and speed is important, SFP should be used for all net sizes.

Speeding up the RSMT generation by orders of magnitude not only accelerates this step of the tool chain but may also improve other steps. For example, the fast run time makes it possible to quickly compute RSMTs during floor planning and placement to provide better routing-cost estimates than the widely used half-perimeter wirelength. Moreover, steps like placement are often run iteratively to optimize the result. By speeding up aspects of these steps, more iterations can be completed in the allotted time, improving the overall result quality. Of course, speeding up the entire tool chain through parallelization would enable much larger improvements. For instance, it might allow chip designers to explore additional dimensions of the design space, thus yielding faster and more energy-efficient chips. We hope our work will inspire others to parallelize and accelerate their VLSI tools to make this a reality.

REFERENCES

[1] Alpert, C.J., Chow, WK., Han, K., Kahng, A., Li, Z., Liu, D., Venkatesh, S. Prim-Dijkstra Revisited: Achieving Superior Timing-driven Routing Trees. 2018 Proceedings of the International Symposium on Physical Design 10-17. 10.1145/3177540.3178239.

[2] Ajwani G., Chu C., and Mak W.K. "FOARS: FLUTE based obstacle-avoiding rectilinear Steiner tree construction." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 2, pp. 194–204. February 2011.

[3] BI1S source code, https://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/RSMT/Robins_BI1S.html, last accessed on 3/15/2022.

[4] Borah M., Owens R.M., and Irwin M.J. "An edge-based heuristic for Steiner routing." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 12, pp. 1563–1568. December 1994.

[5] Z. Cao, T. Jing, J. Xiong, Y. Hu, Z. Feng, L. He, and X. Hong, "Fashion: A Fast and Accurate Solution to Global Routing Problem." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 4, pp. 726-737, April 2008, doi: 10.1109/TCAD.2008.917590.

[6] Chen G. and Young E.F.Y. "SALT: provably good routing topology by a novel steiner shallow-light tree algorithm." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. January 2019.

[7] Chu C. and Wong Y.C. "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 1, pp. 70–83. January 2008.

[8] Cinel S. and Bazlamacci C.F. "A Distributed Heuristic Algorithm for the Rectilinear Steiner Minimal Tree Problem." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 11, pp. 2083–2087. NovEMBER 2008.

[9] FLUTE 3.1 source code, http://home.eng.iastate.edu/~cnchu/flute.html, last accessed on 3/15/2022.

[10] GeoSteiner 5.1 source code, http://www.geosteiner.com/, last accessed on 3/15/2022.

[11] Hanan M. "On Steiner's Problem with Rectilinear Distance." SIAM Journal on Applied Mathematics, vol. 14, no. 2, pp. 255–265. March 1966.

[12] Hwang F. K. "On Steiner Minimal Trees with Rectilinear Distance." SIAM Journal on Applied Mathematics, vol. 30, no. 1, pp. 104–114, Jan. 1976.

[13] ISPD 2008 Global Routing Contest inputs, http://www.ispd.cc/contests/08/ispd08rc.html, last accessed on 3/15/2022.

[14] Jayaraman R. and Rutenbar R.A. "A Parallel Steiner Heuristic for Wirelength Estimation of Large Net Populations." 1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers, pp. 344–347. 1991.

[15] Kahng A.B., Lienig J., Markov I.L., and Hu J. "VLSI Physical Design: From Graph Partitioning to Timing Closure." Springer. 2011. ISBN 978-90-481-9590-9.

[16] Kahng A.B. and Robins G. "A New Class of Iterative Steiner Tree Heuristics with Good Performance." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 11, no. 7, pp. 893–902. July 1992.

[17] Li L., Qian Z., and Young E.F.Y. "Generation of optimal obstacle-avoiding rectilinear Steiner minimum tree." Proceedings of the 2009 IEEE/ACM International Conference on Computer-Aided Design, pp. 21–25. November 2009.

[18] Rojas F. and Meza F. "A Parallel Distributed Genetic Algorithm for the Prize Collecting Steiner Tree Problem." Proceedings of the 2015 International Conference on Computational Science and Computational Intelligence, pp. 643–646. 2015.

[19] SALT source code, https://github.com/chengengjie/salt, last accessed on 3/15/2022.

[20] SFP source code, https://cs.txstate.edu/~burtscher/research/SFP/, last accessed on 3/15/2022.

[21] Warme D.M. "A new exact algorithm for rectilinear Steiner trees." Network Design: Connectivity and Facilities Location, vol. 40, pp. 357–395. 1997.

[22] Warme D.M., Winter P., and Zachariasen M. "Exact Algorithms for Plane Steiner Tree Problems: A Computational Study." Advances in Steiner Trees, vol. 6, pp. 81–116. 2000.

[23] Zachariasen M. "Rectilinear full Steiner tree generation." Networks: An International Journal, vol. 33, no. 2, pp. 125–143. February 1999.

[24] Zachariasen M. and Rohe A. "Rectilinear group Steiner trees and applications in VLSI design." Mathematical Programming, vol. 94, no. 2-3, pp. 407–433. January 2003.

[25] Zhou H. "Efficient Steiner tree construction based on spanning graphs." Proceedings of the 2003 International Symposium on Physical Design, pp. 152–157. April 2003.

[26] Zhou H., Shenoy N., and Nicholls W. "Efficient minimum spanning tree construction without Delaunay triangulation." Proceedings of the 2001 Asia and South Pacific Design Automation Conference, pp. 192–197. 2001.