GPU-Accelerated VLSI Routing using Group Steiner Trees

Venkata Suhas Maringanti
Department of Computer Science
Trinity College Hartford, CT 06106
venkatasuhas.maringanti@trincoll.edu

Basileal Imana
Department of Computer Science
Trinity College Hartford, CT 06106
basileal.imana@trincoll.edu

Peter Yoon
Department of Computer Science
Trinity College Hartford, CT 06106
peter.yoon@trincoll.edu

ABSTRACT
The problem of interconnecting nets with multi-port terminals in VLSI circuits is a direct generalization of the Group Steiner Problem (GSP). The GSP is a combinatorial optimization problem which arises in the routing phase of VLSI circuit design. This problem has been intractable, making it impractical to be used in real-world VLSI applications. This paper presents our work on designing and implementing a parallel approximation algorithm for the GSP based off an existing heuristic on a distributed architecture. Our implementation uses the CUDA-aware MPI approach to compute the approximate minimum-cost Group Steiner tree for several industry-standard VLSI graphs. Our implementation achieves up to 103x speedup compared to the best known serial work for the same graph. We present the speedup results for graphs up to 3k vertices. We also investigate some performance bottleneck issues by analyzing and interpreting the program performance data.

1. INTRODUCTION
A number of optimization problems with different application areas can be modeled by the GSP: given an undirected weighted graph $G = (V, E)$ and a family $N = \{N_1, \ldots, N_k\}$ of k disjoint groups of nodes $N_i \subseteq V$, find a minimum-cost tree which contains at least one node from each group N_i. One of such problems is the global routing phase in VLSI design. The exponential increase in complexity of integrated circuits where tens and thousands of non-overlapping nets may need to be routed simultaneously makes VLSI design a broad area where combinatorial optimization methods can be applied. The problem of interconnecting a net with multi-port terminals is a direct generalization of the GSP.

The advent of modern petascale supercomputing architectures has enabled scientists and engineers to solve several complex problems. Today's supercomputers can not only perform calculations with blazing speed, but also process vast amounts of data in parallel by distributing computing tasks to thousands of processing elements. With portable Application Programming Interfaces (APIs) such as MPI (Message Passing Interface) and CUDA (Compute Unified Device Architecture), researchers can now exploit parallelism to not only solve bigger problems, but also solve more problems in shorter time. This paper presents our work on the design and implementation of a parallel approximation algorithm for the GSP that uses Depth Bounded Steiner Tree Approximation [1]. Our goal was to achieve a better run time to make the heuristic practical for very large scale problems.

DOI: https://doi.org/10.22369/issn.2153-4136/8/1/4

2. A GPU-BASED ALGORITHM
Given an instance of the GSP, our parallel implementation returns a minimum-cost group Steiner tree. Our parallel algorithm follows the following steps in order.

2.1 Metric Closure on GPU
In general, the given graph G may violate the triangle inequality, i.e., there may be edges in G whose cost is greater than the cost of the minimum u-v path in G. An optimal group Steiner tree will contain no such edges, since replacing such edges with the corresponding shortest paths will decrease the total tree cost. Therefore, without loss of generality, we replace G by its Metric Closure. The Metric Closure is defined as the complete graph where the cost of each edge (u, v) is equal to the cost of the minimum u-v path in G. In other words, our first task is to compute the All Pair Shortest Paths (APSP) for the given graph and replace every edge cost with the corresponding minimum u-v path cost. For the APSP, we use a highly efficient CUDA implementation for the Blocked Floyd-Warshall APSP algorithm from [2] on a GPU. After computing the metric closure and replacing the original graph with it, we modify G as follows. We duplicate and replace every port with a new node and add a zero-cost edge between the two. The original port is now a non-port and the newly added node is now a port. An optimal tree in the modified graph G' has the same cost as an optimal tree in the original graph. Hence, this transformation allows us to seek a near-optimal Steiner tree in the original graph.

2.2 Group Steiner Heuristic on CPU
Using G' from the previous step, we now construct a minimum-cost Group Steiner tree by launching multiple processes using CUDA-aware MPI. A *d-star* is defined to be a rooted tree of depth at most d. With every vertex as the potential root r, we follow the following steps in order to construct the final solution.

2.2.1 1-Star
We construct 1-star tree rooted at the root of the optimal solution tree, i.e. a tree of depth 1 where all leaves are ports, one from each group.

2.2.2 Minimum-Norm Partial Star
We then select the intermediate nodes and determine a set of groups that should be connected to each intermediate node. A root, an intermediate node and a set of groups together form a

Partial-Star. Each partial-star is a sub-tree of the solution tree. We compute all such partial stars until all the groups are spanned.

2.2.3 *2-Star*

We combine all the partial-stars computed in the previous step to form the 2-star solution tree for the given vertex.

2.2.4 *Minimum-cost 2-Star*

We then collect all such 2-star solution trees obtained from the previous step and select the one with the minimum cost as the final solution. This is the minimum cost Steiner tree that the algorithm is supposed to output.

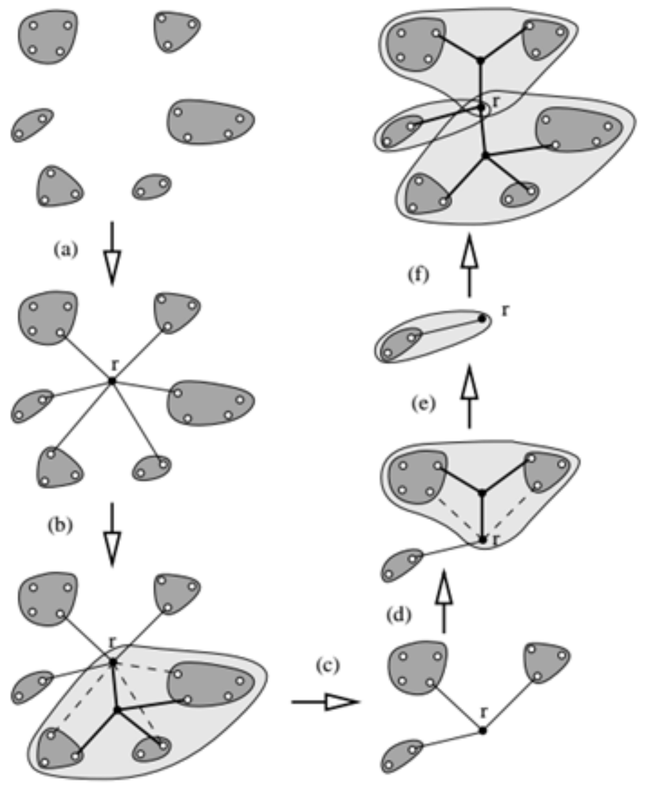

Figure 1. Steps (a) through (f) of the GSP Heuristic [1]

As shown in Figure 1, (a) constructs a rooted *1-star*, i.e. a tree of depth 1 where all leaves are ports, one from each group. A root, an intermediate node and a set of groups together form a *partial-star*. Each minimum-*norm* partial-star is a sub-tree of the solution tree [1]. Steps (c) though (e) compute such partial-stars until all groups are spanned. Step (f) combines all the partial-stars computed in the previous step to form a 2-star tree for the given root *r*. Out of all such 2-star trees obtained from the previous step, the one with the minimum cost is the final solution.

2.3 Work Distribution

Our Hybrid CPU-GPU approach uses CUDA-aware MPI as the standard for launching multiple processes on the Blue Waters supercomputer. In distributing work, the popular master-slave approach was used, wherein process 0 is the "master" process and the remaining ones are "slave" processes. The master performs step 2.1 and then broadcasts the modified graph to all the slaves. Each vertex (a potential root) is then mapped to a slave, whose task is to perform step 2.2 and communicate the result back to the master. After receiving all such results, the master then performs reduction to compute the overall solution.

2.4 NP-Hardness of GSP

The Group Steiner Problem (GSP) is a direct generalization of Classical Steiner Tree Problem, and has been known to be NP-hard. Hence it is not known whether an optimal solution to the GSP can be found by using a polynomial-time algorithm. The GPU-based algorithm as described above is a polynomial-time approximation scheme that efficiently outputs a near-optimal Group Steiner Tree.

The *performance ratio* is defined as the ratio of the approximate cost to the optimal cost for a given instance of the GSP. The Group Steiner Heuristic as described above returns a solution with a performance ratio no more than $2.\left(2+\ln(\frac{k}{2})\right).\sqrt{k}$ where k is the number of groups in the given instance of GSP [1].

The results in Table 1 show the comparison between the best known upper bound of the optimal cost (Opt. Cost) and the approximate cost(Approx. cost) returned by our GPU-based approach. Our results show that the GPU-based approach returns a nearly optimal solution with negligible cost error and a performance ratio within the given upper bound.

3. PERFORMANCE EVALUATION

We ran our performance tests on Blue Waters supercomputer which uses a Cray XE6/XK7 system. We tested both our serial and parallel implementations using several Wire Routing Problem (WRP) instances from industry. The instances are in a widely accepted standard STP format [3]. We compare our approximate solutions for WRP instances with optimal solutions from [4]. Our analysis shows the cost error is less than 1% for all the input graphs that we tested on.

Graph name	Opt. cost	Approx. cost	Error %
wrp3-11	1100361	1100427	0.006
wrp3-39	3900450	3900600	0.004
wrp3-96	96001172	96003009	0.002
wrp3-83	8300906	8302279	0.017

Table 1. Error of approximate cost

The graph below shows a comparison between the running times for the best known serial work [4] and our parallel implementation. We tested on several graph sizes ranging from 128 to 3168 vertices. Our analysis shows that our algorithm achieves speed-ups for bigger graph sizes (>600 vertices), with a maximum speed-up of 103x for the wrp3-83 graph with 3168 vertices.

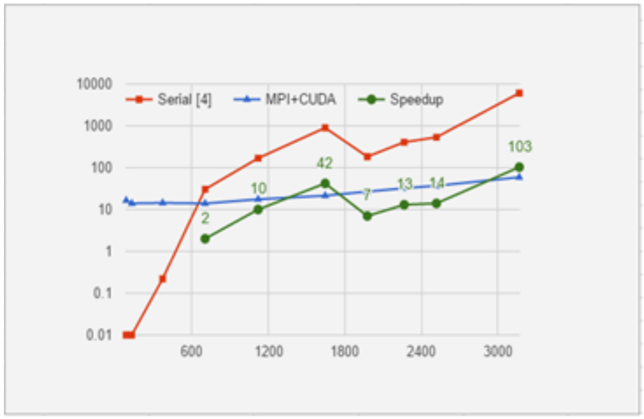

Figure 2. Serial time, parallel time and speedup vs graph size

A common task in HPC is measuring the *scalability* (also referred to as the *scaling efficiency*) of an application. This measurement indicates how efficient an application is when using increasing numbers of parallel processing elements. The graph in figure 3 shows that our problem is highly scalable for a problem size of 2518 vertices (wrp3-96).

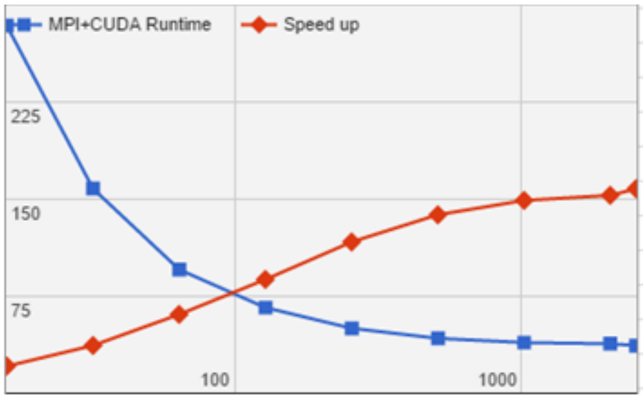

Figure 3. Parallel time and speed-up vs processes

4. CONCLUSION

After careful analysis, we have noticed some subtle yet interesting points about our algorithm. Our algorithm is highly dynamic in the sense that it is not possible to predict the size of the solution at any point before actually computing it. Because of this, the work needed to be done at every step, which is proportional to the output size, cannot be predicted. This means that work distribution is highly irregular and this leads to load imbalance among processes which inhibits performance. This uncertainty also contributes to a lot of irregular memory accesses which is also a performance bottleneck. Our algorithm is also adaptively refined, in the sense that it uses several steps to refine the solution for each vertex and then chooses the best solution among all the vertices. Algorithms that share the same characteristics as ours also share the problems of load imbalance and memory hierarchy.

5. FUTURE WORK

Our implementation suffers from the problems of load imbalance and irregular memory accesses due to the highly dynamic nature of our algorithm. Hence we wish to design a better load balancing mechanism and optimize the memory consumption of our implementation. Future work also includes making modifications to overlap more computation with data-communication.

6. REFLECTIONS

The project described in this paper was Venkata's Blue Waters Student Internship project where he learned to incorporate several principles of computation and high-performance computing into his research. This section presents Venkata's reflections about his internship and the impact that it has had on his current and future academic endeavors: My interest in computer science was ignited right from the introductory courses that I took my freshman year in college. I was fortunate to have received an opportunity to work with Prof. Yoon on this research project right from my freshman summer. At the end of the summer, we had the sequential and parallel versions of the code running on our local cluster. To our surprise, the parallel version was slower than its sequential counterpart in terms of run-time. After thorough investigation we concluded that our implementation had suffered from load balancing and thread divergence issues that hurt the performance a lot. We articulated that significant parts of our algorithm were more suitable to be handled by the CPU than the GPU and hence we started looking into distributed computing architectures like the Blue Waters Supercomputer. I then applied for the Blue Waters Internship Program and was fortunately accepted for the summer after my sophomore year in college. At the 2-week workshop, I learned parts of the C and FORTRAN programming languages in order to learn the basics of the parallel computing libraries OpenMP, CUDA, MPI, and OpenACC. I was taught how to use profiling and debugging tools like CPMAT and TAU. I was also exposed to parallel I/O libraries such as Lustre. Thanks to this experience, I am now confident using the Linux command line to navigate Blue Waters and write basic shell scripts to execute the code for my research. Having learned these skills, I am capable of using supercomputers for my research, which is at the intersection of engineering and computer science. This research experience has given me a glimpse of how computer scientists carry out research that continually shapes the world we live in. I am planning on pursuing a doctorate at a graduate school in the field of computational science, and I believe that these experiences will make me a good candidate in the application process. The Blue Waters Internship is definitely a turning point in my college career and my life in general, and having this opportunity to do cutting-edge research with real-world implications is an invaluable experience.

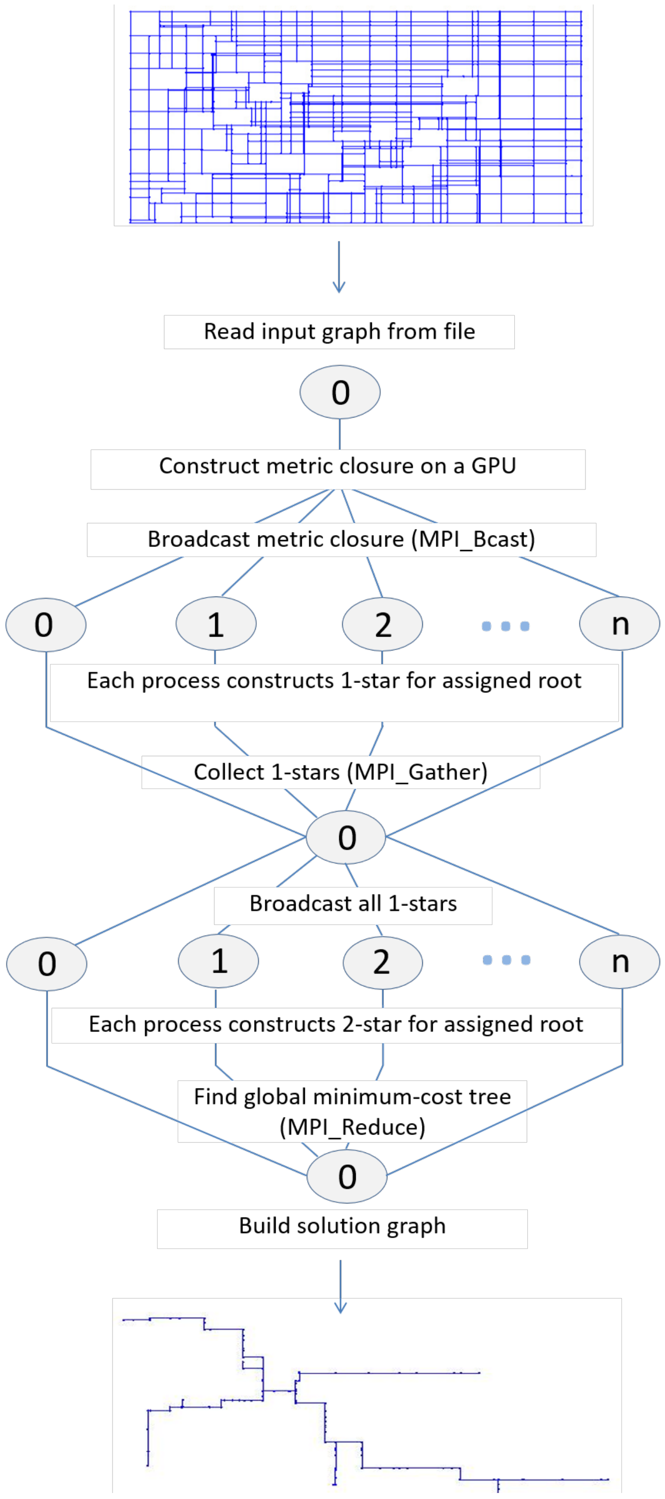
Read input graph from file
0
Construct metric closure on a GPU
Broadcast metric closure (MPI_Bcast)
0
1
2
n
Each process constructs 1-star for assigned root
Collect 1-stars (MPI_Gather)
0
Broadcast all 1-stars
0
1
2
n
Each process constructs 2-star for assigned root
Find global minimum-cost tree (MPI_Reduce)
0
Build solution graph

Figure 4. A graphical flow chart representation of the parallel algorithm.

```
IF master
    $G \leftarrow$ read_graph($filename$)
    $(Mc, P) \leftarrow$ FLOYD_APSP_CUDA($G$) /*get metric closure
    and predecessors matrix*/
ENDIF
BROADCAST($Mc$) /*from master*/
$onestar\_all \leftarrow [\,]$
WHILE there is a remaining root $r$
     $onestar \leftarrow$ build_onestar($Mc, r$)
     GATHER($onestar, onestar\_all$) /*to master*/
ENDWHILE
BROADCAST(onestar_all) //from master
$global\_min \leftarrow \{\,\}$
$local\_min \leftarrow \{\,\}$
WHILE there is a remaining root $r$
    $twostar \leftarrow$ build_twostar($Mc, r, onestar\_all$)
     IF cost($twostar$) < cost($local$_min)
          $local$_min $\leftarrow twostar$
     ENDIF
ENDWHILE
REDUCE($local\_min, global\_min$) /*to master*/
IF master
     build_sol_graph ($global\_min, P, Mc$)
ENDIF
```

Figure 5. A parallel approximation algorithm for GSP.

7. ACKNOWLEDGMENTS

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. We thank the Blue Waters Student Internship Program for providing Venkata with this opportunity. We also thank the Summer Science program at Trinity College, which provided Venkata and Basileal with room and board for the summer.

8. REFERENCES

[1] Helvig C.S., Robins, G. and Zelikovsky, A. New Approximation Algorithms for Routing with Multi-Port Terminals. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *19*(10), 1118-1128.

[2] Lund, B. D., and Smith, J. W. A multi-stage CUDA Kernel for Floyd-Warshall. CoRR abs/1001.4108 (2010).

[3] Koch, T., Martin, A. and Voß, S. SteinLib: an updated library on Steiner tree problems in graphs. in Cheng, X. and Du, D.Z. eds. *Steiner Trees in Industry,* Springer US, Berlin, 2001, 285–326.

[4] Polzin, T., Vahdati, S. The Steiner tree challenge: An updated study, *11th DIMACS Implementation Challenge.* Retrieved July 06, 2015, from Princeton University: http://dimacs11.cs.princeton.edu/papers/PolzinVahdatiDIMACS.pdf.