Parallel VLSI Detailed Routing Using General-Purpose Computing on Graphics Processing Unit

Lalinthip Tangjittaweechai[1], Mongkol Ekpanyapong[1], Kanchana Kanchanasut[2]
[1]Microelectronics and Embedded Systems
[2]Computer Science department
Asian Institute of Technology, Pathumthani, Thailand
st113130@ait.ac.th, mongkol@ait.ac.th, kk@cs.ait.ac.th

Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology
Georgia, USA
limsk@ece.gatech.edu

Adriano Tavares
Department of Industrial Electronics
University of Minho
Guimarães, Portugal
yannitavares@gmail.com

Prabhas Chongstitvatana
Computer Engineering department
Chulalongkorn University
Bangkok, Thailand
prabhas@chula.ac.th

***Abstract*—Parallelization of VLSI routing algorithms is one of the challenging problems in VLSI physical design. This is due to a large number of nets as well as the shared routing resources that result in data dependency among concurrent tasks. In this paper, VLSI Maze routing using GPGPU has been proposed to enable runtime performance improvement. We report up to 3x performance gain with an average of 25% runtime performance improvement from VLSI Maze routing using CPU. The routing qualities including wirelength and overflow are better among all benchmarks comparing with CPU baseline. The solutions also scale well when the size of the problem increases.**

Keywords- VLSI Routing; GPGPU; CUDA

I. Introduction

IC chips are the integration of hundreds of millions of transistors or more into a single die. One of the VLSI physical design process is the VLSI routing. The objective of VLSI routing is to complete all circuit connections by using the shortest possible wirelength, least total overflows, in the shortest amount of time. Several million transistors are contained in a VLSI chip and tens of thousands of nets have to be routed to complete the circuit connection. Moreover, there may be several hundreds of possible routes for each net and this makes the routing problem intractable.

VLSI routing can be classified into two groups: sequential routing and parallel routing. Many routing algorithms are based on sequential approaches running on CPU. In 1961, Lee proposed Lee's maze router [1] which is an algorithm for routing a two terminal net on a grid and is guaranteed to find a shortest path, if the path exists. For routing multi-terminal net, Steiner tree based algorithm [1] is frequently used. In addition, routing problem can be changed to a 0/1 integer programming formulation called Integer Programming Based Approach [1].

For the concurrent approach, Tai-Hsuan Wu, Azadeh Davoodi and Jeffrey T. Linderoth research [2] proposed a routing algorithm which uses an integer programming formulation running on CPU. The routing problem is divided into sub-problems affecting the quality of the output and the execution time. Wen-Hao Liu et al. [3] proposed a parallel multi-threaded collision-aware global router based on Maze Routing Algorithm using a heuristic collision-prevention approach. In this proposed algorithm, task-based concurrency strategy is used. The proposed algorithm uses a heuristic to solve this race condition.

To accelerate the computation time, General-Purpose computing on Graphics Processing Units (GPGPU) [4] is introduced to speed up the computation time by running processes in parallel. To do so, Compute Unified Device Architecture (CUDA) [5] which is a programming framework is employed. In this paper, VLSI Maze routing algorithm is modified to run using GPGPU computing paradigm. In section II, some preliminaries concepts are described. Our implementation of routing algorithm base on GPGPU and some optimizations are described in section III. The experimental results are shown in section IV. Finally, the conclusion is explained in section V.

II. Preliminaries

A. *Routing Model and Matrics*

The routing problem is to find a path connecting the vertices in a grid graph G(V,E) [6]. The chip is partitioned into rectangular regions called global routing cells represented by vertex v_i. The connection between v_i and v_j is represented by edge e_{ij}. The maximum number of net that can be routed over the edge e_{ij} is represented by c_{ij} and d_{ij} is defined the number of nets which are routed on edge e_{ij}. There are three important routing metrics generally used which are the total overflows (the summation of the overflow of all edges in the grid. Edge e_{ij} is overflown if $d_{ij}>c_{ij}$ and the overflow of edge e_{ij} is equal to d_{ij}-c_{ij}), the total wirelength (sum of wirelength that is used to connect all nets to complete the circuit) and the total execution time (the time that is used to run the complete program).

B. *Routing a Multi-Terminal Net*

The input of routing problem is multi-terminal nets and each multi-terminal net has their pins or terminals. For routing a multi-terminal net, a multi-terminal net has to be separated into two terminal nets and the two-terminal nets are routed. Then, all paths are combined to get the complete path of the multi-terminal net.

C. *GPGPU and CUDA*

GPGPU is a technique using GPU which are microprocessors attached in graphic cards to execute operations in parallel. This technique is used to solve general purpose problems on graphics hardware (GPU). CUDA is a parallel computing programming model increasing computing performance by running programs on GPU introduced by NVIDIA. A parallel part of applications is executed in a function called kernel. CPU invokes the kernel and the kernel is launched with *N* blocks and each block has *M* threads on the GPU. There are Streaming Multiprocessors (SMs) inside GPU for executing blocks of threads. Each SM has Streaming Processors (SPs) executing individual thread and shared memory.

III. Methodology

A. *VLSI Routing Algorithm*

Our VLSI routing algorithm is based on Maze routing algorithm proposed by [6]. The runtime breakdown of ibm10 benchmark of the program is shown in Fig. 1. The time that is used to route all nets without rip up and reroute phase takes about 68% of the program's total time and the time that is used in rip up and reroute phase consumes about 30% of the total time.

The baseline algorithm is partitioned in such a way that some parts are operated on GPU while other parts are executed on CPU. The flowchart of our routing algorithm is shown in Fig. 2 where shaded area is running on GPU. The Maze routing algorithm starts by converting multi-terminal nets into two-terminal net using prim's algorithm. After that the algorithm will route one two-terminal net at a time until there is no two-terminal net to be routed. Once all nets have been routed, the total wirelength and overflow will be calculated. Then, in the rip up and reroute phase, all nets that pass through overflowed area will be ripped up. The algorithm operates on one routing track (routing edge) at a time. Once the overflown edge is found, all nets that pass through that edge have to be ripped and rerouted. The algorithm will continue until there is no improvement for 20 times of rip up and reroute cycle.

B. *CUDA Implementation of Prim's Algorithms*

The prim's algorithm [7] is introduced to separate a multi-terminal net into two-terminal nets. The objective of prim's

This work was supported by Thailand Research Fund, Office of Higher Education Commission, and School of Engineering, Asian Institute of Technology under contract number MRG5380293.

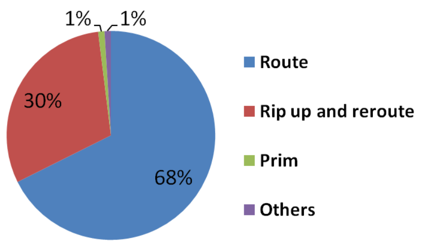


Figure 1. CPU runtime breakdown.

algorithm is to form a tree connecting all vertices by using shortest total wirelength. Therefore, we will know which pairs of terminal should be connected.

There are many multi-terminal nets which have to be routed to complete the circuit. All multi-terminal nets have to invoke the prim's algorithm. Since, there is no data dependency among different multi-terminal net, all multi-terminal nets can be processed at the same time. In our implementation, one thread is used to extract one multi-terminal net into two-terminal nets. Hence, the algorithm can run in parallel. Here, the total number of threads is equal to the number of multi-terminal nets.

C. *Implementation of Routing Two-Terminal Net*

The algorithm to route a two-terminal net is based on Maze routing algorithm. To route a two-terminal net, one thread is represented by one node in the grid graph. The routing algorithm is based on weighted cost computation when the algorithm tries to find the shortest weighted path from the source to the sink. At the beginning, all threads are set the cost value of each node to a maximum number except the source node which is set to zero. Each thread calculates its cost from the cost of its neighbors and chooses the lowest cost among four neighbors (top, bottom, left, right). A thread will update its cost if the lowest computed cost is less than its current cost. When threads update the cost, threads also set its parent for retracing purpose (tracing to the source). The cost of each neighbor is calculated by using (1).

$$\text{Cost} = \text{Manhattan distance (current node to sink)} + \text{Manhattan distance (current node to source)} + \text{ROUTECOST}*(\text{neighbor point's cost} + \text{capacity}) \qquad (1)$$

The cost is calculated by using the Manhattan distances from the current node to the sink node, the Manhattan distances from the current node to the source node and the multiplication of ROUTECOST with the cost of the neighbor plus the capacity of the edge. The ROUTECOST is a constant value set to 20 similar to [6]. All threads look at its neighbors and update its cost concurrently.

Fig. 3 shows an example of cost calculation. In the Figure, M represents the maximum value, S represents the source node and T represents the sink node. At the beginning the green threads update its cost from the source node as shown in Fig. 3(a). After that, the blue thread calculates the cost from

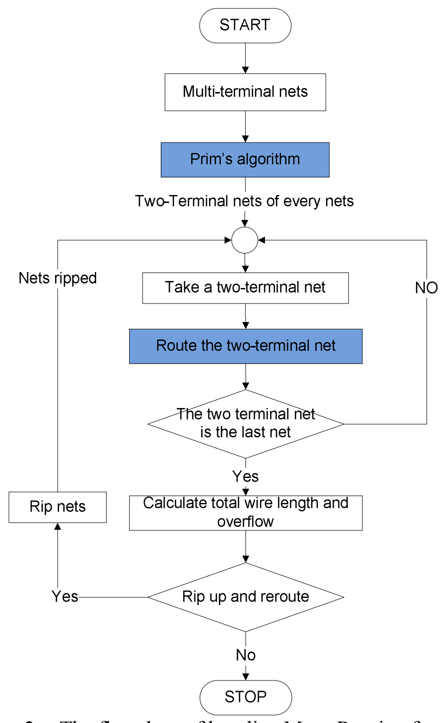
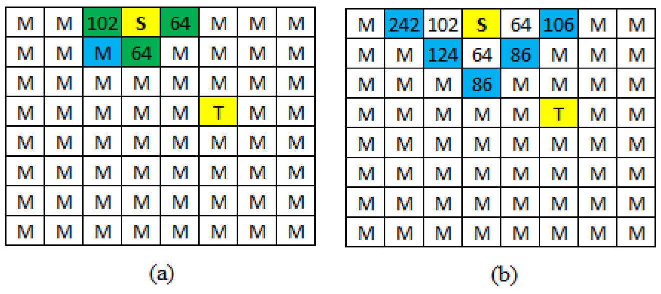

START
Multi-terminal nets
Prim's algorithm
Two-Terminal nets of every nets
Nets ripped
Take a two-terminal net
NO
Route the two-terminal net
The two terminal net is the last net
Yes
Rip nets
Calculate total wire length and overflow
Yes
Rip up and reroute
No
STOP

Figure 2. The flowchart of baseline Maze Routing framework.



M	M	102	S	64	M	M	M
M	M	M	64	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	T	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M

(a)

M	242	102	S	64	106	M	M
M	M	124	64	86	M	M	M
M	M	M	86	M	M	M	M
M	M	M	M	M	T	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M

(b)

Figure 3. (a) Calculating the cost from its neighbors. (b) The latest updated. nodes

the cost of its neighbors which are 102, 64, and two maximum values by using Eq. *(1)*. After all the green threads updated its cost, the blue threads in Fig. 3(b) will update its cost in the next iteration.

In this algorithm, we allow the cost to be updated as long as the new cost is lower than the current value. Our algorithm is stopped when there is no updating or the cost of the sink is less than or equal to the minimum cost of the latest updated nodes. This means that the sink cannot update its cost. Also, this algorithm can be stopped when the minimum cost of the latest updated nodes is greater than or equal to the cost of a node which is already in the routing path in the same multi-terminal net. The parallel reduction algorithm [8] is used to find the minimum cost in parallel. The parallel reduction uses tree-based approach within each block. One thread is used to compare two numbers. The parallel reduction reduces the number to be compared into half in every step and continues until the number is equal to 1.

D. Optimization

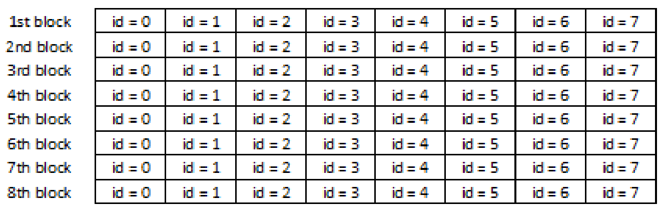
This section explains some optimization techniques on CUDA to reduce the execution time. This algorithm creates y blocks and each block has x threads when y is equal to the height of the grid and x is equal to the width of the grid. If grid size is 8*8, the threads and blocks are represented in Fig. 4.

The first problems which is time-consuming is redundant data loading, each thread has to load the information of its neighbors to calculate the cost propagated from a memory called global memory. Fig. 5 shows that two threads load the same data represented by the brown node which is redundant loading.

In GPGPU, there are memories called shared memory. Thread can communicate with the other threads in the same block by using shared memory. Shared memory resides inside each SM on GPU. Because of this, the latency to access shared memory is lower than acquiring data from the global memory. Each block has its own shared memory and threads cannot access shared memory belonging to the other blocks. To reduce the execution time of the redundant loading, each thread loads its data to the shared memory as shown in Fig. 6. Threads will load the data of the left and right nodes from the shared memory (instead of the global memory) but they still have to load the data of the top and bottom nodes from the global memory.

The second problem is to find the lowest cost of the latest updated node for stop conditions. To perform partial reduction, the data size of grid height*grid width* size_of_integer have to be used for data computation. For a large grid graph, this could result in large memory bandwidth. To address this problem, shared memory is also used. When a thread updates its cost value, it will also update the cost to the shared memory. After all threads in the block finish updating its cost value, partial reduction is called at the block level on the shared memory to reduce memory bandwidth to the global memory. Then, the lowest cost of the block is written to the global memory (grid height). Hence, the data which have to be loaded from the global memory to find the minimum cost is reduced to the number of grid height*size_of_integer instead of the number of grid height*grid width*size_of_integer.

GPU has the limitation on the number of thread per block. For example, GeForce GTX 480 [9] cannot launch more than 1024 threads per block. In our implementation, the number of thread in a block is equal to the number of nodes in a row. If the maximum number of thread per block is 4 and the grid width is 8, two iterations are required as shown in Fig. 7. The 1st thread calculates the 1st node. After finish calculating the 1st node, the 1st thread then calculates the 5th node. At the same time, the 2nd node calculates the 2nd node and, then the 6th node respectively.



1st block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
2nd block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
3rd block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
4th block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
5th block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
6th block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
7th block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
8th block	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7

Figure 4. Designing the threads and blocks in a grid.

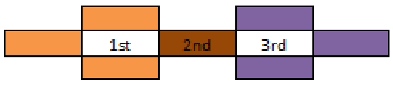


Figure 5. Redundant data loading from global memory.

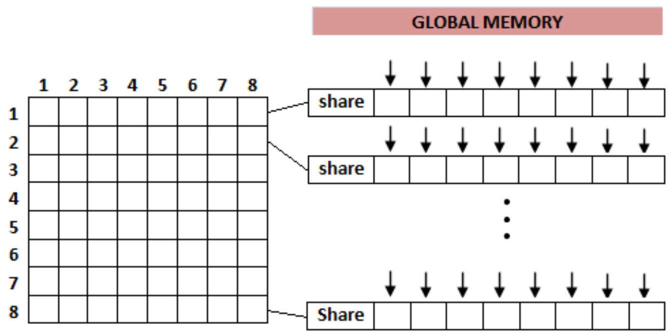


Figure 6. Each thread loads data from global memories to shared memories.

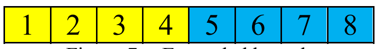


Figure 7. Expanded kernel.

IV. Experimental Results

In the experiment, we use ASUS GeForce GTX 480 which has 480 Cores for the graphics card and Intel(R) Core2 Duo CPU E7500 @ 2.93GHz for the CPU. Ten benchmarks from ISPD98/IBM [10] are used. Two algorithms are compared in this section: The CPU algorithm is based on maze routing from [6] named CPU. We implement CUDA program of our algorithm named GPU:Maze Table I illustrates the comparison of total wirelength, overflow and the execution time including rip up and reroute phase.

Our CUDA program uses less total wirelength (reduces 6% on average), gets less total overflows (reduces 36% on average). CPU algorithm uses a queue to choose the next node to expand. Due to the fact that our algorithm did not use queue like CPU to expand to the next node, our algorithm can expand to more paths and has more opportunities to find the better paths. Our program is faster than the CPU with up to three fold runtime reduction with the average of 25% for the large benchmarks.

V. Conclusion

In this paper, we employ the concept of GPGPU computation's paradigm to improve the routing quality of VLSI Maze routing algorithm. The experimental results show that we can improve the routing quality in case of total wirelength and total overflows. For the execution time, the CUDA program can speed up the execution time up to 3x times the sequential program with the average of 25% runtime reduction.

References

[1] N. A. Sherwani, Algorithms for VLSI Physical Design Automation, 3rd ed., KLUWER ACADEMIC: Boston, 2002.

[2] T.-H. Wu, A. Davoodi and Jeffrey T. Linderoth, "A Parallel Integer Programming Approach to Global Routing," Design Automation Conference - DAC, pp. 194-199, 2010.

[3] W.-H. Liu, W.-. Kao, Y.-L. Li and K.-Y. Chao, "Multi-threaded collision-aware global routing with bounded-length maze routing," Design Automation Conference - DAC , pp. 200-205, 2010.

[4] GPGPU: General-Purpose computing on graphics processing units, URL: http://gpgpu.org

[5] D. Kirk and W.-M. Hwu, Programming Massively Parallel Processors, SANTA CLARA: Calif, 2010.

[6] R. Kastner, E. Bozorgzadeh and M. Sarrafzadeh, "Pattern Routing: Use and Theory for Increasing Predictability and Avoiding Coupling," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, July 2001.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd ed., The MIT Press, Sep 2001.

[8] Parallel reduction, URL: http://developer.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[9] NVIDIA, Geforce gtx 480, URL: http:/www.nvidia.com.

[10] Benchmarks,ISPD98/IBM,URL: http://cseweb.ucsd.edu/~kastner/labyrinth_vault/benchmarks/index.html

TABLE I. The Comparison Of Total Wirelength, Overflow and The Execution Time Including Rip and Reroute

Benchmarks	Grid Size	Number of Nets	Wirelength		Overflow		time (s)	
			CPU	GPU:Maze	CPU	GPU:Maze	CPU	GPU:Maze
ibm01	64 * 64	11507	76474	71360	355	327	15	15
ibm02	80 * 64	19291	195530	177385	229	104	22	23
ibm03	80 * 64	21621	178696	159018	118	100	23	22
ibm04	96 * 64	26163	194890	181541	975	727	46	34
ibm05	128 * 64	27777	420583	419172	0	0	29	33
ibm06	128 * 64	33354	318270	307553	163	143	51	42
ibm07	192 * 64	44394	429129	409132	634	259	129	60
ibm08	192 * 64	47944	458355	434140	548	240	135	84
ibm09	256 * 64	50393	468335	448979	360	316	178	76
ibm10	256 * 64	64227	648354	620089	446	90	276	84