GPU-based Line Probing Techniques for Mikami Routing Algorithm

Chiu-Yi Chan† Jiun-Li Lin⋆ Lung-Sheng Chien◇
s996004@mail.yzu.edu.tw p76004148@mail.ncku.edu.tw lungshengchien@gmail.com
Tsung-Yi Ho⋆ Yi-Yu Liu†
tyho@csie.ncku.edu.tw yyliu@saturn.yzu.edu.tw

† Department of Computer Science and Engineering, Yuan Ze University, Taiwan 320, R.O.C.

⋆ Institute of Computer Science and Information Engineering, National Cheng Kung University, Taiwan 701, R.O.C.

◇ Department of Mathematics, National Tsing Hua University, Taiwan 300, R.O.C.

Abstract— Graphic processing unit (GPU), which contains hundreds of processing cores, is becoming a popular device for high performance computation in multi-core era. With strictly computation regularity characteristic, specific algorithms are key challenges for performance speed-up. In this paper, we propose a parallel CUDA-Mikami routing algorithm on NVIDIA's GPU. A 32-bit routing grid encoding is proposed to simplify wire intersection identification and wire direction recognition. Furthermore, thread-level and warp-level line probing techniques are proposed for vertical and horizontal routings, respectively. The experimental results indicate that the run-time efficiency is promising as compared to traditional CPU-version algorithms.

I. Introduction

Multi-core processing is an industrial trend for high performance computing since the processor clock frequency has reached its physical limit. As the number of computing cores increases, parallel algorithm is the key to fully utilize the computing power of a multi-core system. Many parallel algorithms have been developed based on various parallel computing architectures [1]. Graphic processing unit (GPU) is traditionally used for graphics and image computations with high computation demands and low data dependency characteristics. In recent years, GPU integrates hundreds of processing cores to fulfill the needs for high definition images and videos. High data throughput is achieved by GPU since many operations can be simultaneously performed on massive homogeneous cores. With the parallel computation capability on GPU, lengthy run-time of some applications can be dramatically improved. Therefore, GPU becomes an attractive device to improve computation performance of various applications.

In modern VLSI physical design, millions of components are required to be connected in a single chip. Routing, which determines wire segments of all interconnections without violating design rules, has become a time-consuming step due to the huge number of instances and chip dimension. Many wire routing algorithms are proposed for various routing objectives and design styles. Among them, maze routing is one of the most important routing algorithms for wire length minimization [2]. However, the run-time complexity and memory requirement of maze routing algorithm result in serious problems in a large-scale VLSI design. Mikami and Tabuchi propose a run-time efficient routing algorithm based on line probing technique [3]. The proposed algorithm guarantees to obtain a routing result with minimum number of bends. In this work, we are motivated to utilize NVIDIA's GPU computing power for Mikami routing algorithm optimization. To the best of our knowledge, this is the first paper which proposes a GPU-compliant line probing technique for wire routing.

The rest of this paper is organized as follows. We briefly introduce conventional routing algorithms and NVIDIA's GPU architecture in Section II. Our CUDA-Mikami routing algorithm is proposed in Section III. The experimental results and the race condition issue are discussed in Section IV. Finally, we conclude this paper and point out further research directions in Section V.

II. Background

A. Conventional Routing Algorithms

Maze routing, also known as Lee's algorithm, is a grid-based path-finding technique based on breadth-first search (BFS) algorithm [2]. With the nature of BFS, maze routing algorithm guarantees to obtain a routing result with minimum wire length from source-point to target-point. There are two major stages in maze routing: wave propagation and back trace. In wave-propagation stage, adjacent grids are incremented by 1 from source-point to target-point. In back-trace stage, a routing path is obtained from target-point to source-point. The drawback of a two-dimensional maze routing algorithm is the quadratic run-time complexity and memory storage. A huge number of grid size could significantly impact the efficiency of maze routing algorithm.

Mikami routing algorithm employs line probing technique to speed-up the run time of BFS wave propagation. In line probing stage, with setting both source-point and target-point as base grids, Mikami routing generates two level-1 cross lines (one horizontal wire and one vertical wire) for each base grid. All the grids on the level-l cross lines will become new base grids in next level. Mikami routing iteratively generates cross lines from base grids until there exists cross-line intersection from source and target points. After that, back traces are performed from the intersection point to source-point and target-point, respectively. The actual routing path is completed after back traces. Mikami routing effectively reduces the run time of maze routing algorithm.

In summary, maze routing guarantees for routing result with shortest wire length and Mikami algorithm guarantees for minimum bend count. The search space and time complexity of maze and Mikami algorithms are $O(MN)$ and $O(L)$, respectively, where M and N are the horizontal and vertical grid size, and L is the number of Mikami routing levels. The graph grids data dependency of Mikami algorithm is lower than that of maze algorithm, since Mikami algorithm performs cross line propagation and maze algorithm performs grid-by-grid wave propagation.

B. *Graphic Processing Unit*

GPU computing and GPGPU (General-Purpose Computation on Graphics Processing Units) are the techniques of using GPU to perform general purpose computation. GPU is traditionally used for graphics and image computations. Nowadays, GPU collaborates with CPU (host) in a heterogeneous manner for various applications. GPU computing is challenging due to the intrinsic graphic operations in GPU. NVIDIA develops "Compute Unified Device Architecture (CUDA)" parallel programming model to ease the implementation challenges of general purposed GPU computing [4, 5]. Fermi is the latest CUDA programming model [6]. There are plenty of intrinsic functions supported in Fermi to enhance the performance of GPU computing.

In CUDA, thread is a fine-grained pseudo GPU processing unit for each data-parallel and data-independent operation. Each thread is assigned to a GPU streaming processor (SP) for actual execution. Since each GPU contains hundreds of SPs, a GPU program achieves high throughput if most SPs are executing data-parallel tasks simultaneously. A warp, which contains 32 threads, is a scheduling unit on NVIDIA's GPU. Each warp can be scheduled to a streaming multiprocessor (SM) for execution if all required data is ready. Taking NVIDIA GTX 580 as an example, there are 16 SMs and each SM contains 32 SPs. Hence, there are total 512 SPs in a GTX 580. To fully utilize a GPU, the most important issue is to increase the number of threads for SMs execution. We will use GTX 580 as our implementation platform in this work.

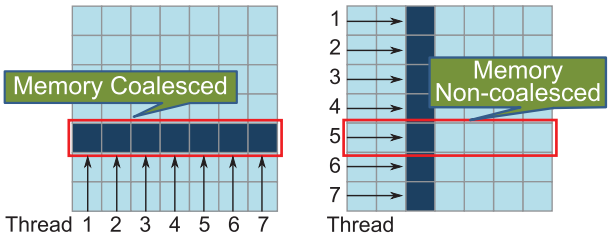

Fig. 1. Memory coalesced and non-coalesced phenomena.

In addition, memory access is another key issue for GPU programming. Since a GPU has its own individual memory hierarchy, data transfer is required between host and GPU memories. To reduce the data transfer overhead between two memories, we need to prevent unnecessary communications between host and GPU. In GTX 580, each off-chip memory access fetches 128-byte data into a cache line. Assume that there are 32 active threads within a warp and each thread requires a 4-byte data. If all 32 required data are not in the same cache line, there are 32 cache misses (off-chip memory accesses) within a warp. This phenomenon is so-called non-coalesced memory access. If all 32 required data are in the same cache line, there is only 1 cache miss (off-chip memory access) within a warp. This phenomenon is so-called coalesced memory access. A warp with non-coalesced memory access will be deferred for a long memory access latency. If all warps are waiting for memory accesses (inactive warps), the SM utilization as well as the GPU data throughput will be drastically reduced. Therefore, coalesced memory access within a warp significantly improves GPU performance. Figure 1 indicates that 7 vertical threads shares one memory access (cache line). However, 7 horizontal threads require 7 memory accesses.

III. CUDA-Mikami Routing Algorithm

We propose a parallelized CUDA-Mikami routing algorithm on NVIDIA's GPU. Our algorithm is composed of three stages: preparation stage, CUDA-Mikami stage, and back-trace stage. In the preparation stage, we generate cross lines from source-point and target-point. All grids on the lines are taken as the initial base grids for CUDA-Mikami stage. In CUDA-Mikami stage, vertical and horizontal segments are generated from base grids by using our proposed line probing techniques. All grids on the generated segments are taken as the base grids for next level line probing. The line probings are iteratively performed until an intersection point has been found. Finally, CPU performs back-trace to determine the routing path. Figure 2 illustrates the overall routing algorithm.

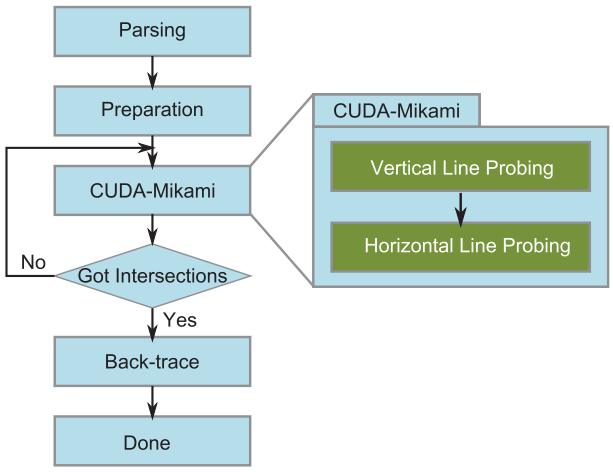

Fig. 2. GPU-based Mikami routing algorithm.

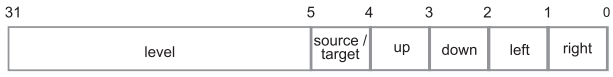

Fig. 3. 32-bit routing grid encoding.

A. Routing Grid Encoding

To efficiently utilize the precious memory resource in a GPU, we encode routing level, line source/target, and line direction into a 32-bit integer routing grid. The routing level records the sequence number of line segments. Source/target bit distinguishes the origin of line segments. The direction field indicates line probing directions. Since there are four different directions, we use 0001(1), 0010(2), 0100(4), and 1000(8) to represent rightward, leftward, downward, and upward directions, respectively. Figure 3 illustrates our proposed encoding scheme.

With our proposed encoding scheme, we can easily identify wire intersection by taking XOR operation on the source/target bit of two adjacent grids. The wire intersection type can be recognized according to the direction bits from the grids by Equation 1. *SourceDir* and *TargetDir* represent source-point and target-point directions, respectively. Figure 4 draws all wire intersection types.

$$\text{intersection type} = (SourceDir + TargetDir)\%8 \quad (1)$$

B. Line Probing

Line probing marks routing grids according to our proposed encoding scheme. CUDA-Mikami algorithm iteratively generates horizontal and vertical wire segments for line probing. All grids on the generated wires are base grids for next level vertical and horizontal line generations. Since the memory-access patterns of horizontal and vertical lines are different, we separate them into two

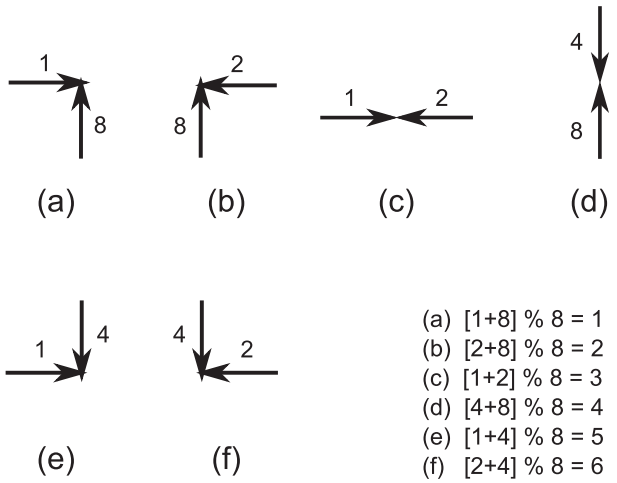

Fig. 4. Wire intersection types.

individual line probings and propose different algorithms to support coalesced memory accesses.

As mentioned in Section II, memory accesses between adjacent routing grids can be easily coalesced in vertical line probing. Therefore, we propose a thread-level approach, which generates a vertical line by one thread. Each thread checks the routability of the corresponding vertical line and marks all routable routing grids. All marked grids in current level are base grids for next level line probings. A thread stops line probing on the first encountered un-routable grid, which contains a boundary grid, an obstacle grid, and a marked grid.

In horizontal line probing, non-coalesced memory accesses occur if we assign a thread to generate one horizontal line. As illustrated in Figure 1, 32 off-chip memory accesses per warp result in huge performance degradation. To avoid non-coalesced memory access and to improve cache hit rate, we propose a warp-level line probing technique, which generates a horizontal line by one warp. Before discussing horizontal line probing using the warp-level approach, we first introduce two CUDA intrinsic functions _ballot() and _ffs() [4]. Function _ballot() evaluates a predication for all 32 threads within a warp and returns an integer. Each thread asserts a bit to 1 if and only if the thread-corresponding predication is evaluated to non-zero. The predication is user-defined and we take the predication as the routability of one grid in this work. Function _ffs() is same to common Linux ffs() function. The _ffs() returns the position of the first asserted bit from least significant bit of an integer. The two intrinsic functions are employed within a warp for routability checking and routable grid marking. Hence, all memory accesses are coalesced within each warp.

In our warp-level horizontal line probing, all 32 threads within a warp handles its corresponding routing grid simultaneously. We use _ballot() to check the routabilities of 32 adjacent grids. The corresponding bit of each thread

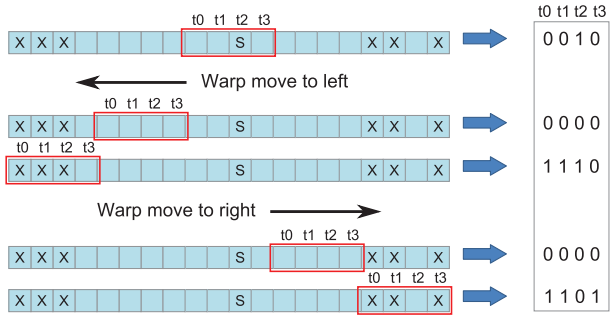

Fig. 5. Example of horizontal line probing.

TABLE I
RUN-TIME COMPARISONS

Benchmarks	*CPU (msec)*	*GPU (msec)*		*Ratio (%)*	
		No warp	Warp	No warp	Warp
ben2210x2220	75.93	81.53	58.56	93.1	129.7
ben2120x2120	84.40	73.04	49.50	115.5	170.5
ben2345x2134	75.83	75.87	60.19	99.9	126.0
ben2400x2100	80.98	60.69	59.96	133.4	135.1
ben2450x2420	142.90	102.03	77.39	140.1	184.6
ben2700x2525	150.18	91.30	81.03	164.5	185.4
ben2425x2900	169.21	83.65	81.76	202.3	207.0
ben2813x2836	179.46	123.14	94.24	145.7	190.4
ben2800x2550	180.81	88.85	88.54	203.5	204.2
ben2819x2953	200.65	103.80	97.33	193.3	206.2
Average	134.04	88.39	74.85	149.1	173.9

TABLE II
BEND-COUNT COMPARISONS

Benchmarks	*CPU*	*GPU*	*Difference*	*Ratio (%)*
ben2210x2220	345	343	2	0.6
ben2120x2120	358	357	1	0.3
ben2345x2134	301	301	0	0.0
ben2400x2100	253	253	0	0.0
ben2450x2420	190	192	2	1.1
ben2700x2525	226	226	0	0.0
ben2425x2900	232	232	0	0.0
ben2813x2836	267	265	2	0.7
ben2800x2550	188	190	2	1.1
ben2819x2953	222	222	0	0.0
Average				0.4

are de-asserted to 0 for routable grid and asserted to 1 for un-routable grid. The routability result obtained by _ballot() is then used for routable grid marking. There are two horizontal routing directions, leftward and rightward. For leftward routing, the _ffs() of routability result specifies the first un-routable grid. For rightward routing, the _ffs() of *bitwise reversed* routability result specifies the first un-routable grid. After that, the warp marks all routable grids between the base grid and the first un-routable grid. Each warp continues to handle next 32 horizontal routing grids unless an un-routable grid is found. Figure 5 illustrates our horizontal line probing. For simplicity, we assume there are four threads in a warp. Symbol 'S' and 'X' represent the base grid and obstacle grids, respectively. In Figure 5, the warp checks grid routability and marks routable grids. The horizontal wire segment is completed when the first un-routable grid is found in both directions.

IV. Experimental Result and Race Condition Issue

We perform the routing experiments on a Linux-based machine with Intel Core i7 CPU 930 @ 2.80GHz, 8G memory and NVIDIA GeForce GTX 580. Our CUDA-Mikami router is compiled with gcc 4.4.5 and nvcc 4.0.

We randomly generate 10 benchmark designs with thousands of obstacles to evaluate the efficiency of our CUDA-Mikami router. We also implement Mikami router in CPU-version for comparisons. The major difference between CUDA-Mikami and CPU-version Mikami router is the approach of line probing. In CPU-version, we sequentially generate all the lines one-by-one as the conventional Mikami algorithm proposed; In CUDA-Mikami, we use massive threads to generate all the lines simultaneously on NVIDIA's GPU.

Table I lists our experimental results. We compare the results of CUDA-Mikami, denoted *GPU*, to the result of CPU Mikami, denoted *CPU*. Columns *No warp* and *Warp* indicate the results of CUDA-Mikami without and with warp-level line probing technique during horizontal line probing, respectively. The ratios of CPU to GPU run-time improvement are listed in column *Ratio*. According to Table I, memory coalesced line probing achieves 173.9% run-time improvement while memory non-coalesced line probing achieves only 149.1% run-time improvement.

Our next experiment compares the routing quality of CPU and GPU Mikami algorithms. The total number of routing bends are listed in Table II. From Table II, we notice that the routing bend number of CUDA-Mikami are slightly greater than that of CPU Mikami. To understand the bend-count differences between the two routers, we analyze the routing results and identify a race condition issue during line probing.

Since vertical and horizontal line probings employ different line generating techniques, race condition may occur when the two line probings are performed simultaneously. In horizontal line probing, we use function _ballot() to check the routability and function _ffs() to identify the first un-routable grid. All routable grids, between the base grid and the first un-routable grid, will be marked in this level. If an unmarked and routable grid is read by one vertical thread between horizontal routability checking and grid marking, race condition occurs since the grid can be marked by both vertical and horizontal line probings. Taking Figure 6 as an example, CUDA-Mikami marks source bits of all level-1 routable grids to 1 and 0 from source-point and target-point, respectively. After level-1 cross line generation, all the blue grids become base grids. In level-2 cross line generation, assume

TABLE III
RESULTS OF SEPARATED LINE PROBINGS

Benchmarks	*CPU (msec)*	*GPU (msec)*		*Ratio (%)*	
		Combined VH	*Separated VH*	*CPU to GPU*	*Performance Degradation*
ben2210x2220	75.93	58.56	67.93	111.8	16.0
ben2120x2120	84.40	49.50	85.37	98.9	72.5
ben2345x2134	75.83	60.19	64.06	118.4	6.4
ben2400x2100	80.98	59.96	64.33	125.9	7.3
ben2450x2420	142.90	77.39	95.00	150.4	22.7
ben2700x2525	150.18	81.03	104.08	144.3	28.5
ben2425x2900	169.21	81.76	112.42	150.5	37.5
ben2813x2836	179.46	94.24	108.64	165.2	15.3
ben2800x2550	180.81	88.54	109.08	165.8	23.2
ben2819x2953	200.65	97.33	126.51	158.6	30.0
Average	134.04	74.85	93.74	139.0	25.9

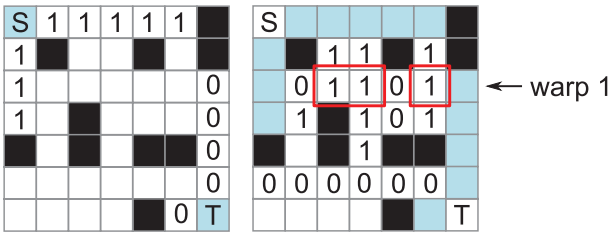

Fig. 6. Race condition between line probings.

that the horizontal routability checking is performed by warp 1 and at the same time line probings are performed by vertical threads. Three unmarked and routable grids, labeled in red boxes, are simultaneously accessed by three vertical line probing threads and one horizontal line probing warp. As a result, all the three routable grids can be marked by either vertical line or horizontal line. Since each grid can be marked for once, the horizontal line is segmented by a vertical line when the vertical line marks the grid, as illustrated in Figure 6. Otherwise, the vertical line is segmented by a horizontal line. This race condition incurs a routing result with unnecessary routing bends.

To avoid the race condition, we separate horizontal and vertical line probings in a sequential manner and get exactly same bend count with CPU Mikami. The results of race-condition-free router are listed in Table III. Columns *Combined VH* and *Separated VH* indicate the routing results with and without race condition, respectively. Column *CPU to GPU* indicates the run-time ratio of CPU Mikami to race-condition-free CUDA-Mikami. Column *Performance Degradation* indicates the performance degradation of separated line probings. From Table III, we can find out that the race-condition-free CUDA-Mikami router has 26% performance degradation as compared to the original CUDA-Mikami router due to the reduced number of parallel threads. The overall run-time reduction as compared to CPU Mikami router becomes 139%.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the GPU-based line probing issues. We propose a 32-bit routing grid encoding and a simple equation to efficiently recognize intersection wire directions. Both thread-level and warp-level line probing techniques are proposed to avoid non-coalesced memory access for vertical and horizontal routings, respectively. The experimental results indicate that there is 139% run-time efficiency in average as compared to traditional CPU-version algorithms. Currently, we are extending our single-net CUDA-Mikami router to a multiple-net global router. Since we do not utilize share memory to reduce the number of global memory accesses, there is much room for further performance improvement.

REFERENCES

[1] Prithviraj Banerjee, "Parallel Algorithms for VLSI Computer-Aided Design, 1st edition," *Prentice-Hall*, 1994

[2] Lee, C. Y. "An Algorithm for Path Connections and Its Applications," *IRE Transactions on Electronic Computers*, EC-10 (2): 346365, 1961.

[3] K. Mikami, K.Tabuchi, "A computer program for optimal routing of printed circuit connectors," *Proceedings of IFIP*, H47:1475-1478, 1968.

[4] "NVIDIA CUDA Programming Guide 4.0," *NVIDIA*

[5] "NVIDIA CUDA Best Practices Guide 4.0," *NVIDIA*

[6] Fermi architecture, *http://www.nvidia.com/object/fermi_architecture.html.*