FPGA-Accelerated Maze Routing Kernel for VLSI Designs

Xun Jiang[1], Jiarui Wang[2], Yibo Lin[2], Zhongfeng Wang[1]
[1]School of Electronic Science and Engineering, Nanjing University, Nanjing, China
[2]CECA, CS Department, Peking University, Beijing, China
jiangx@smail.nju.edu.cn, {jiaruiwang, yibolin}@pku.edu.cn, zfwang@nju.edu.cn

2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC) | 978-1-6654-2135-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASP-DAC52403.2022.9712533

***Abstract*—Detailed routing for large-scale integrated circuits (ICs) is time-consuming. It needs to finish the wiring for millions of nets and handle complicated design rules. Due to the heterogeneity of net sizes, the greedy nature of the backbone maze routing, and interdependent workloads, accelerating detailed routing with parallelization is rather challenging. In this paper, we propose a FPGA-based implementation to accelerate the maze routing kernels in a most recent detailed router. Experimental results demonstrate that batched maze routing kernel is 3.1× speedup on FPGA. Besides, our design gets deterministic results and has less than 1% quality degradation on ISPD 2018 contest benchmarks [1].**

***Index Terms*—maze routing, detailed routing, FPGA acceleration, VLSI designs**

I. Introduction

Routing is known as the most time-consuming step in very-large-scale-integration (VLSI) back-end design flow [2] [3]. Modern detailed routing needs to process millions of nets and handle complicated design rules, significantly slowing down the design closure. The main-stream detailed routing engines leverage maze routing as the kernel path finder and are optimized for multi-threaded CPU platforms [4], [5]. With the tremendous increase in problem sizes and complexity, parallelization on CPU can hardly bring any more benefits. Thus, new paradigms leveraging heterogeneous computing platforms are desired.

Recent detailed routers like TritonRoute [5] and Dr.CU 2.0 [6] have demonstrated promising performance on industrial benchmarks from ISPD 2018/2019 contests. They mainly follow the nature of sequential routing algorithms, i.e., iteratively invoking the path finder (e.g., maze routing) to route each net and adopting the rip-up and reroute scheme to avoid routing congestions. To leverage the multi-threading resources on CPU, Dr.CU [4] proposes a batch-based parallelization scheme, by extracting a batch of independent nets from the layout and performing maze routing simultaneously. While each batch can contain thousands of nets, the benefits from CPU multi-threading saturate at 8–16 cores [6].

To further speedup routing algorithms, the literature has explored acceleration on heterogeneous platforms like GPU and FPGA. For VLSI designs, only GPU acceleration for global routing has been investigated due to the large scale and complicated design rules in the detailed routing problems. Han et al develop a GPU-accelerated global router [7], achieving 2.5–3.9× speedup with 2.5% wirelength degradation. They later improve the scheduling strategy for net-level concurrency and the GPU implementation of the maze routing algorithm [8], eventually achieving 4× speedup with 1% wirelength degradation compared with the academic router NTHU-Route 2.0 [9]. However, most of the studies on GPU acceleration suffer from data dependency of routing algorithm, heavy data transfer and synchronization overhead. As far as we know, only Korolija et al [10] propose an FPGA-accelerated implementation of the VPR router for FPGA routing, but ending up with 4–6× slower than running on Intel Core i5. Therefore, it is challenge to design high-performance accelerator of routing algorithm for the studies on FPGA acceleration.

In this paper, we propose the first implementation of maze routing kernel on FPGA for VLSI detailed routing. The main challenges are summarized as follows:

- Data dependency within maze routing on a single net.
- Heterogeneity in the size of nets.
- A large number of random memory access.

The main contributions of our work are summarized as follows:

- We design a highly efficient data structure and high-performance implementation for maze routing on FPGA.
- We propose a scalable scheduler and the interconnect network for multiple maze routing processing elements on FPGA.
- We achieve up to 3.1× speedup of the batched maze routing with less than 1% wirelength degradation on AWS FPGA cloud servers compared with Dr.CU 2.0 [6].

We believe the realistic performance results reported in this work will shed light on future research on routing acceleration.

The rest of the paper is organized as follows. Section II describes the background and motivation; Section III explains the detailed implementation; Section IV demonstrates the system design, hardware implementation, and parallelism analysis; Section V demonstrates the results; Section VI concludes the paper.

II. Preliminaries

This section will review the background on the detailed routing algorithm and basic FPGA acceleration.

A. Detailed Routing Algorithm

Detailed routing algorithm is critical to routing closure. It optimizes the global routing results at a lower level of granularity. The routing resources contain a stack of metal layers. On each layer, wire segments, which are usually named as *tracks*, run either horizontally or vertically. *Vias* are introduced to connect the wires on the adjacent metal layers.

During routing, we usually abstract the routing grids to a 3-D grid graph $G(V, E)$. V denotes the set of vertices at the location of *tracks* separated by fixed distances. E denotes the set of edges representing the wire segments or vias to connect the vertices on the grid graph.

The state-of-the-art detailed routers like Dr.CU 2.0 [6] adopt the rip-up and reroute (RRR) scheme to optimize the routing quality with maze routing as the kernel path finder for each net. Firstly, it assigns nets into batches and employs a scheduler to determine the execution order of nets. These batches must run in sequential, while nets within a batch can be executed in parallel. Within each iteration, if DRC violations occur when routing a net, the net will be rip-up and wait for rerouting in the next iteration. In general, the routing will gradually converge with more iterations. In such a sequential routing scheme, the efficiency and quality of the batched maze routing kernel are critical to the overall routing performance. In this work, we focus on accelerating such a kernel on a CPU-FPGA platform.

978-1-6654-2135-5/22/$31.00 ©2022 IEEE

B. FPGA Acceleration

FPGA is a flexible computing platform that enables user-defined hardware architectures and software programming models. A FPGA device contains distributed LUTs, SRAMs, DSPs and other computing resources, which provide dedicated parallelism for specific computing. Traditional workloads running on CPUs need to be carefully designed to fully utilize the massive parallelism of FPGA. There are three main techniques to design a high-performance accelerator on FPGA: 1) pipeline the computing task with custom logic, 2) improve the ratio of data reuse on FPGA, and 3) use efficient data structure and memory access behavior to saturate the bandwidth of DRAM.

Furthermore, most FPGA devices also provide PCIe interfaces for communication with the host CPU, which enables us to leverage the power of the heterogeneous computing platforms. Meanwhile, since CPU and FPGA do not share the memory, minimizing the overhead of data communication is important to the overall performance. Coarse grained control signals, data blocks, and interrupts are often adopted to optimize the system performance.

III. Algorithms

In this section, we introduce the algorithm for our hardware-optimized maze routing and the detailed implementation of the algorithm on FPGA.

A. Hardware-Optimized Maze Routing

Dr.CU 2.0 [4] has implemented an efficient maze routing algorithm on the multi-core CPU server. It utilizes the priority queue and some heuristic methods to improve the quality and speed. However, the typical maze routing algorithm should be modified to adapt the property of FPGA.

The goal of maze routing is to connect all pins on the nets with minimal cost. In the actual computing stage, the nets will first abstracted as grid graphs. Each pin p_n is equivalent to a set of vertices S_n located on its position. The priority queue Q, the temporary cost set T, and the vertices' state set F used in the hardware-optimized maze routing algorithm are also friendly to implemented on hardware.

At the beginning of the Algorithm 1, T, F, and Q should be initialized to infinity, false, and empty in line 2. Then, the vertices with the start pin are pushed into Q, whose costs are written to T. The main body of searching path consists of two nested loops on the grid graph. The inner one is the loop of vertices, that comes to an end at the time of reaching an unconnected pin. The outer one is the loop of pins, that needs to connect all pin on the grid graph. Therefore, the path P with minimal cost is a set of partial paths in a connected graph.

From lines 11 to 24, there are operations used to construct the total path with many partial paths. In Line 11, `GetPartialPath` is used to get the result generated using the vertex and its predecessor by one inner loop. In Line 12, the connected pin is removed from the unconnected pin set $\{p_n\}$, because the partial path of it has been connected to the total path. From Line 14 to 17, the temporary cost of the vertex in the partial path is set to zero, and these vertices with zero cost are pushed into the priority queue. This operation enables the next partial path can be connected to any vertex in the total path to get the minimal cost. From Line 18 to 21, the vertices of the searched pin are pushed to the priority queue and temporary cost set, because there are no connections between vertices on the same pin. Line 22 means the state of vertices will be cleaned at the end of the inner loop, which ensures the cost of a temporary total path is always minimal. Line 25 and 26 mean that if the vertex has been searched in the same inner loop, the computing will be ignored to reduce redundant computing. From Line 28 to 37, there is the basic expansion of the vertex to its neighbors by comparing the new cost of the vertex with the old one. The critical part of maze routing is preserving the search frontier (priority queue Q) and reducing redundant work to improve work efficiency.

Algorithm 1 Hardware-Optimized Maze Routing.

Input: A local grid graph $G(V, E)$, N sets of vertices $\{S_n\}$ related to pins $\{p_n\}$.
Output: Path P with minimum total cost.

```
1: A priority queue Q, a set of temporary cost T, a set of vertices' state F.
2: t ← ∞, ∀t ∈ T; f ← false, ∀f ∈ F; Q ← ∅
3: for v ∈ S_0 do                                   ▷ Initialize start pin
4:     T[v] ← v.cost
5:     Q.push(v)
6: end for
7: Remove p_0 from {p_n}
8: while {p_n} ≠ ∅ do                               ▷ Loop of pins
9:     while Q ≠ ∅ do                               ▷ Loop of vertices
10:        u ← Q.pop()
11:        if u.pin ∈ {p_n} then
12:            GetPartialPath(P, u.pin)
13:            Remove u.pin from {p_n}
14:            for v ∈ P.partial do
15:                T[v] ← 0
16:                Q.push(v)
17:            end for                              ▷ Reset vertices in partial path
18:            for v ∈ S_{u.pin} do
19:                T[v] ← v.cost
20:                Q.push(v)
21:            end for                              ▷ Push vertices from searched pin
22:            f ← false, ∀f ∈ F
23:            Break
24:        end if
25:        if F[u] = true then
26:            Continue                             ▷ Reduce redundant work
27:        else
28:            F[u] ← true
29:            for v ∈ u.neighbors do
30:                u.penalty ← PenaltyFunc(u, v)
31:                v.cost ← u.cost + (u → v).w + u.penalty
32:                v.length ← LenFunc(u, v)
33:                if v.cost < T[v] and v ≠ u.pred then
34:                    T[v] ← v.cost
35:                    Q.push(v)
36:                end if
37:            end for
38:        end if
39:    end while
40: end while
```

The function `PenaltyFunc`(u, v) and `LenFunc`(u, v) utilize the same heuristic methods as Dr.CU 2.0 [6]. But the whole calculation of the result is separated on CPU and FPGA to take advantage of the bandwidth and the hardware resource on FPGA.

For optimization on hardware, $G(V, E)$, $\{S_n\}$ and $\{p_n\}$ are stored in DRAM on board, but Q, T, and F are implemented using BRAM on chip. The operations on $G(V, E)$, $\{S_n\}$ and $\{p_n\}$ refers to accessing large-capacity DRAM, which is large enough to store these data. The operations on Q, T, and F refers to accessing BRAM, which is tightly coupled with logic for computing with high-level parallelism. These components are orchestrated to complete the maze routing on hardware.

B. Data Structure for Accelerator

In the implementation of maze routing on CPU, the data structure is represented by C++ class. However, this format is unsuitable to be directly used by an accelerator on FPGA, which favors a more regular data structure to fully utilize bandwidth and parallel computing components.

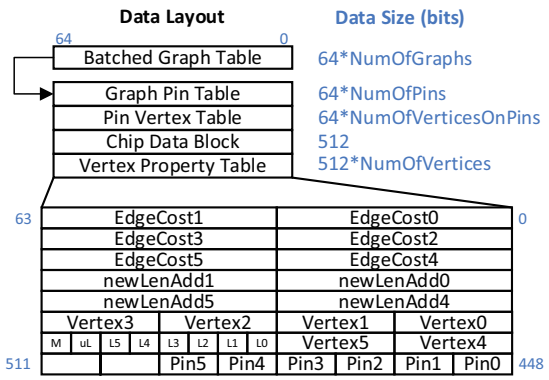


Fig. 1: Data Layout in FPGA Memory.

Because the grid graph is a regular cube grid that each vertex on it has just six adjacent vertices, each entry of the adjacent array has the same bit width. As shown in Fig.1, the memory space of the graph on FPGA is divided into four sections: *Graph Pin Table*, *Pin Vertex Table*, *Chip Data Block*, and *Vertex Property Table*. The entry of *Graph Pin Table* is each pin's bias address and the number of vertices with it. The entry of *Pin Vertex Table* is the vertex and its cost with the same pin index. The entry of *Chip Data Block* includes some data for each layer. The entry of *Vertex Property Table* is the six adjacent vertices of one vertex and their costs, along with other data to computing length and temporary cost.

For maze routing implemented on FPGA, the *Vertex Property Table* is accessed with the highest frequency, which dominates the performance of memory access. So, every entry of it is packed into 512 bits to adapt the width of AXI-4 bus and fully utilize the bandwidth of memory. The experimental result shows the benefit of this design.

In Fig. 1, **L** represents layer index for each adjacent vertex. **uL** represents layer index for original vertex. **M**(4 bits) represents minAreaFixable(1 bit), which indicates if the length of original vertex can be fixed with post routing operations, and areOvlpVertex(2 bits), which indicates zero penalty at the two via directions. **newLenAdd** represents the additional length, which is used in `LenFunc`(u, v) to compute the new length of adjacent vertex.

IV. Implementation

A. An Overview of FPGA Acceleration Framework

The FPGA acceleration framework is shown in Fig. 2. A Graph Dispatcher is designed to receive commands from the host, schedule graphs processed on FPGA, and monitor the status of maze routing processing elements (PEs). Maze Routing PEs Cluster are used to fetch graph data from FPGA memory independently to process routing tasks in parallel.

B. Maze Routing Processing Element

The Maze Routing Processing Element in Fig. 3 is composed of Tri-State Priority Queue, Controller, and Executor. All components work together to complete the maze routing task and store the path with minimal cost in FPGA memory.

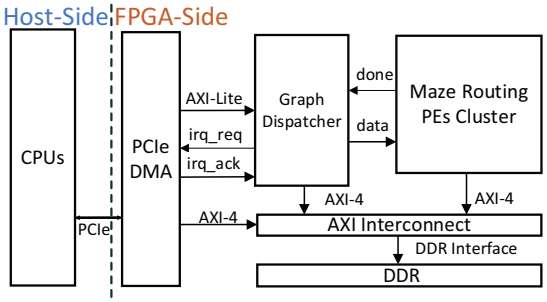


Fig. 2: The FPGA Acceleration Framework on AWS Elastic Compute Cloud (EC2) [11].

1) Tri-State Priority Queue: The backbone architecture of Tri-State Priority Queue is P-Heap [12], which is similar to the structure of minority heap. The priority queue is targeted to sort vertices by temporary cost in ascending order. Due to the structure of pipeline between the adjacent layers of the heap, the time complexity of all operations in the priority queue is reduced to *O(1)*. However, Tri-State Priority Queue has improvements compared to P-Heap [12] and Korolija & Stojilović's work [10].

First, our work employs three operations, which are push, pop, and predict, compared to the traditional two operations in a priority queue. The state transform diagram is shown in Fig. 4. Different from push and pop operations, predict operation only reads the top data from the priority queue, then sends the vertex to Controller with the tag of predict.

Second, the usage of LUTs only increases by the order of log(n) compared to Korolija & Stojilović's work [10], which shows high scalability for massive parallel PEs.

The benefit of prediction is demonstrated on the timeline in Fig. 5. In the maze routing algorithm, vertex with minimal cost will be popped from the priority queue at each iteration to ensure optimality. In a normal situation, only when the total adjacent vertices of the last vertex have been pushed into the priority queue, the next vertex will be popped from the priority queue. Therefore, there is a strict global data dependency between the operations of the priority queue, which is hard to handle on CPU. In this design, a predictive policy is employed to fetch the next vertex's data from memory before the end of the last one. From the structure of Maze Routing Processing Element in Fig. 3, when LSU (Load Store Unit) operates one predictive vertex, the other parts of Executor and Priority Queue will operate on the last vertex. The two periods can be overlapped to hidden the long latency of accessing data from DRAM, as a result, to shorten the total time of one iteration of searching vertex. According to the experimental results of predictive policy, the probability of the second-minimal vertex to be the minimal one at the next iteration is over 80%. Therefore, the predictive policy is beneficial for the implementation of maze routing algorithm with just a little incremental cost of hardware consumption.

In Fig. 5, **PO** represents pop operation in Priority Queue. **PR** represents predict operation in Priority Queue. **CH** represents checking vertex in Controller. **NL** represents skipping loading data from memory in Executor. **LOAD** represents loading data from memory in Executor. **EXEC** represents computing and updating in Executor. **PUSH** represents push operation in Priority Queue. **WAIT** represents the stalls in the processing pipeline.

2) Controller: Controller is composed of Vertex Operation Code Table, Closed Set, and Pin State Table. This module needs to communicate with Graph Dispatcher and schedule different operations of the vertex in a Maze Routing Processing Elements. The scheduling policy in this module can reduce redundant computing and ensure

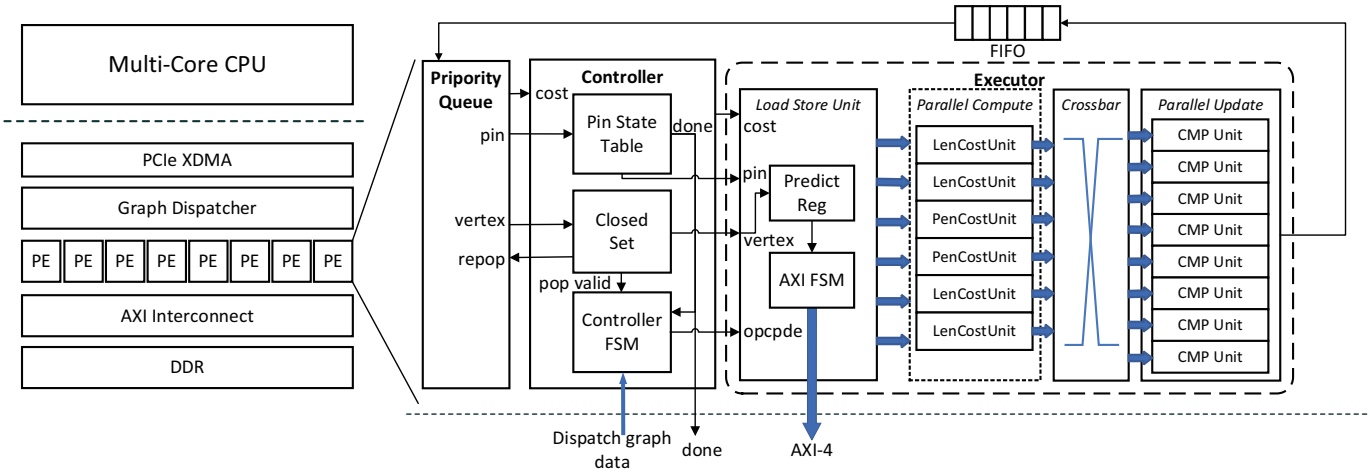


Fig. 3: Structure of Maze Routing Processing Element.

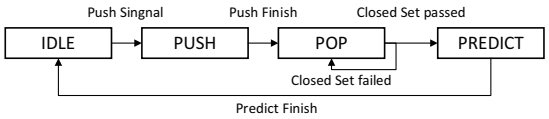


Fig. 4: State transform diagram in Priority Queue.

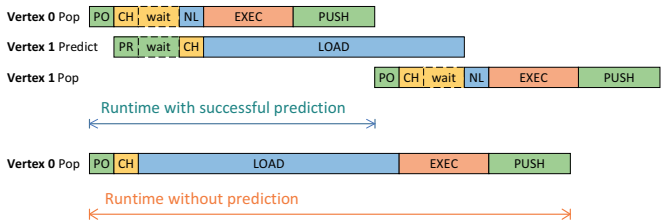


Fig. 5: Impact of Prediction on Runtime.

the optimality of the algorithm.

Vertex Operation Code Table: Different from a simple graph algorithm, maze routing has to handle a more complex schedule policy of vertices. We design a set of vertex operation codes for hardware implementation.

The process of maze routing can be decoupled into two loops of iteration in Fig.6. The inner loop is iteration over vertices, in which vertices are being searched until the vertex belongs to a new pin is searched. The outer loop is iteration over pins, in which pins are being searched until all pins are searched.

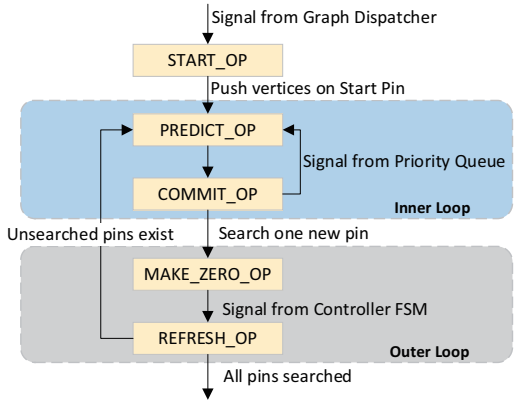


Fig. 6: Operation flow of Controller.

These vertex operation codes will control the behavior of Executor.

1) `START_OP`: Executor should fetch vertices that belong to the start pin, then push them to the priority queue.
2) `COMMIT_OP`: Executor should search the neighbors of committed vertex, then push them to the priority queue if their cost is minimal.
3) `PREDICT_OP`: Executor should search the neighbor of the predicted vertex but just store the information.
4) `MAKE_ZERO_OP`: When one pin is searched in Controller, this opcode will be sent to Executor. This operation will be triggered at the ending of one inner iteration. Executor makes the cost of all vertices on the path searched in this inner loop zero and pushes these vertices with zero cost into the priority queue.
5) `REFRESH_OP`: After the execution of `MAKE_ZERO_OP`, Executor will push all vertices that belong to the searched pin with original cost into the priority queue.

Closed Set: Closed Set is a table used to store the status of vertices, where each entry indicated one vertex's status. In the Closed Set, the true status of vertex means it has been searched during the inner loop of searching one pin, but the false status of vertex means it has not been searched in the inner loop.

Pin State Table: Pin State Table is used to record the status of the pin and the number of vertices related to this pin. When all pin has been searched, Pin State Table will send done signal to Graph Dispatcher.

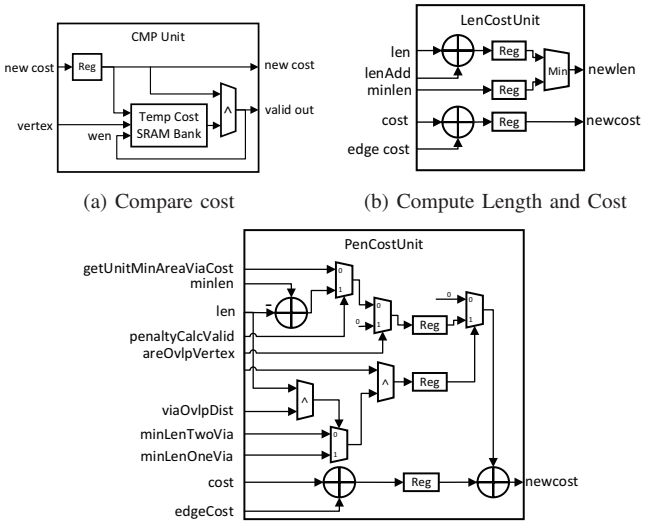
3) Executor: Executor is composed of Load Store Unit, Parallel Computing Unit, Crossbar, and Parallel Updating Unit. The module executes vertex operation codes sent by Controller to access data from off-chip memory, compute new vertices' cost and length, and reconstruct searched paths with minimal total cost. The architecture of Executor is designed for parallel processing of vertices.

Load Store Unit: The Load Store Unit(LSU) consists of registers for vertices' information, the logic for prediction, the logic for reconstructing path with minimal cost, and logic for memory access.

Parallel Computing Unit: The Parallel Computing Unit(PCU) is composed of four *LenCostUnit*s and two *PenCostUnit*s in Fig. 7, which are designed for computing cost, penalty, and length of new vertices adjacent to the last popped vertex.

Crossbar: The Crossbar is an `6-in-8-out` multiplex to dispatch vertices from Parallel Computing Unit to Parallel Updating Unit. The main architecture of Crossbar is 8 `6-in-1-out` arbiters and logic for broadcasting vertices to different arbiters with the lower 3 bits.

Parallel Updating Unit: The Parallel Comparing Unit in Fig. 7 consists of 8 same Comparing Units to fully utilize the parallelism of adjacent vertices of a single vertex.



(a) Compare cost

(b) Compute Length and Cost

(c) Compute Penalty and Cost

Fig. 7: Computing Units in PPU and PCU.

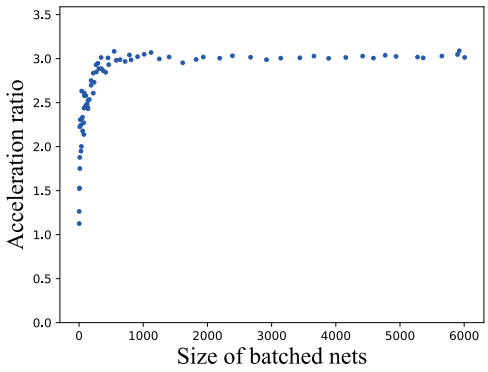


Fig. 8: Acceleration ratio at different batch size compared with 8-thread CPU.

V. Experimental Results

The acceleration framework is developed in Verilog and validated on an Amazon AWS f1.2xlarge instance [11], which has a Linux server with 8-core Intel Xeon CPU E5-2686 v4 @ 2.30GHz with 122GB Memory and the Xilinx Ultrascale+ VU9P FPGA. There are 1.2M LUTs, 76 Mb of Block RAM, and 270 Mb of Ultra RAM on this FPGA device. The hardware design is synthesized by using Vivado 2020.2. We configured the hardware with one cluster of 1, 2, 4, 8, or 12 PEs for the test with a targeted frequency of 125MHz. Dr.CU 2.0 [6] runs on the same AWS server as the baseline.

A. Batched Maze Routing Acceleration

Because of the benchmarks of ISPD2018 and ISPD2019 are both executed by the same maze routing kernel, we just use ISPD2018 benchmark to justify the effectiveness of FPGA acceleration. Batched maze routing kernel is executed on a large batch of graphs in the detailed routing stage. There are almost 1,000 $\sim$ 10,000 graphs (nets) in one batch without data dependency between any two graphs. From the distribution of graphs' sizes in ISPD2018 contest benchmark [1] and the capacity of PE, the small graphs (number of vertices less than 2^{16}, taking more than 90% among all graphs) are accelerated on FPGA.

Fig. 8 shows the acceleration ratio of batched maze routing kernel on FPGA compared with CPUs. The test6 of ISPD2018 contest benchmark [1] is used to show the acceleration of batched maze routing kernel. In Fig. 8, the implementation of batched maze routing kernel on FPGA with 12 PEs is 3.1$\times$ faster than the state-of-the-art detailed router Dr.CU 2.0 [6] running with 8 threads when the batch size is larger than 200. When batch size is within the range of 0 $\sim$ 200, the acceleration ratio is still above 2.0$\times$.

To verify the routing quality of the FPGA-accelerated maze routing kernel, the results of accelerator are dumped and further processed in the software framework of Dr.CU 2.0 [6]. For batched maze routing, the workload is composed of kernels with various batch sizes, the number of PEs is configured to 12, and the number of CPU threads is configured to 8. In TABLE I, the quality degradation of FPGA-Accelerated maze routing is less than 1%. The degradation is caused by omitting the method of expanding searching space in Dr.CU 2.0 [6] for the implementation on FPGA. The detailed routing stage is composed of multiple iterations, where the first iteration is the most time-consuming according to Dr.CU [4]. With the time breakdown of detailed routing in Dr.CU [4], 37.3% of the total runtime is taken by maze routing and the other part is mainly the time of generating routing graphs. Therefore, the Runtime column in TABLE I refers to the batched maze routing in the first iteration of the detailed routing, which is reasonable to demonstrate the promising speedup of maze routing kernel on FPGA.

With the same configuration of the accelerator, we also compare the runtime of the batched maze routing with prediction policy and without it. In TABLE I, the runtime of the test6 on FPGA with prediction policy is 27ms. However, the runtime of the same test without prediction policy is 29ms. Therefore, the prediction policy accounts for 7% performance gain for the batched maze routing.

B. Scalability of Multiple PEs

In Fig. 9, our design shows promising scalability compared with Dr.CU 2.0 [6]. With the number of PEs is 4, 8, or 12, the performance of FPGA is 1.2$\times$, 2.2$\times$, or 3.1$\times$ higher than 8-thread CPU execution mode. The runtime of batched maze routing kernel with 8 threads on CPU server is regarded as baseline runtime, because it has almost reached the peak performance on a multi-core CPU server form Fig. 9. The gain of speedup on multi-core CPU server decreases drastically with 4 threads and beyond, and saturates at 8 threads. The reasons probably come from massive random memory access and synchronization overhead with the number of threads increasing.

The performance of parallel PEs on FPGA increases almost linearly, when the number increases form 1 to 12. We cannot further increase the number of PEs due to the limitation of the FPGA device. From the potential growth of the acceleration ratio, we believe the performance will continue to increase. Considering all PEs just share one independent DDR4 channel, the scalability is remarkable with the compact data structure and concurrent memory access pattern.

Besides, the speed of running maze routing on FPGA is always higher than that on CPUs with the same number of PEs and threads. Especially, when the number of PEs is one, the acceleration ratio is still 1.9$\times$ compared with one CPU thread. Considering the frequency of CPU is 2.30 GHz, which is almost 20$\times$ of the frequency on FPGA (i.e., 125MHz), the custom logic on FPGA shows remarkable performance by leveraging internal parallelism of maze routing and designing high-performance architecture.

C. Consumption of Hardware Resource

The hardware resource consumption of this design is shown in Table II. *PE* is configured with 16-bit vertices and 6-bit pins. `Top(12)` is the top-level module implemented on FPGA with 12 *PE*s. Considering the resource consumption of `Top(12)`, the number of *PE*s can still be increased to boost performance.

TABLE I: Comparison of performance between Dr.CU 2.0 [6] and our FPGA acceleration to complete batched maze routing kernel with ISPD 2018 [1] contest benchmarks.

		Basic Cost		Non-preferred usage					Design Rule Violations			ISPD'18 quality score	Runtime (s)	Speed-up
		WL	#vias	Out-of-guide		Off-track		Wrong-way	Short area	#min area	#spacing			
				WL	#vias	WL	#vias	WL						
ours	test1	430051	32311	2130	425	393	0	5699	0	0	4	290099	2	2.0×
	test2	7819979	325756	38789	6488	5233	0	53452	56000	0	44	4685023	11	2.2×
	test3	8705500	318581	71369	6483	6038	0	60080	36661000	0	93	5291929	11	2.1×
	test4	26365425	734234	388747	29561	17916	0	207602	10351764	184	603	15808947	18	2.4×
	test5	27697175	968543	151866	22360	6586	2	81751	2503540	324	337	16406741	19	2.7×
	test6	35565380	1485088	242069	36889	18077	16	127438	483600	492	492	21666362	27	2.7×
	test7	64954693	2410170	404656	56403	34885	0	200889	7647444	614	287	38523171	44	2.8×
	test8	65251816	2420576	394414	55187	34802	0	196869	6582412	684	325	38717711	43	2.8×
	test9	54533301	2418298	358613	56331	28358	0	191474	654400	980	240	33342024	43	2.9×
	test10	67873499	2603982	1078699	100097	42014	0	250404	100966300	967	960	42820500	49	2.7×
Dr.CU 2.0 [6]	test1	433584	32394	2172	445	419	0	5853	14400	0	2	291305	4	1.0×
	test2	7832656	325674	41036	6542	5291	0	54119	19600	0	57	4700581	24	1.0×
	test3	8718263	318303	65586	6602	5990	0	60553	37074200	0	96	5295331	23	1.0×
	test4	26419880	729197	388623	29618	17721	0	205951	8418136	90	619	15761113	43	1.0×
	test5	27802051	965546	150636	21840	5833	3	77203	2505800	148	361	16370540	52	1.0×
	test6	35703514	1480728	242099	36492	16544	16	119149	598000	271	549	21636717	73	1.0×
	test7	65173353	2402487	402055	55583	32575	0	186348	6855672	343	199	38408621	121	1.0×
	test8	65468212	2412163	402983	54459	32411	0	182755	7223072	392	189	38595625	120	1.0×
	test9	54759612	2410625	358136	55296	26348	0	177596	508000	491	115	33114609	123	1.0×
	test10	68098159	2595706	1091463	99121	40703	0	237319	110771500	619	958	42861890	132	1.0×

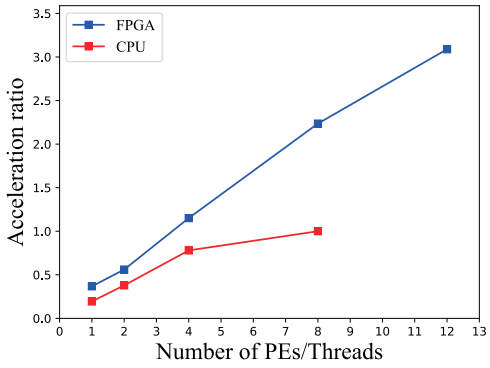


Fig. 9: Acceleration ratio at different numbers of PEs/threads (batch size = 5923).

TABLE II: Hardware resource consumption on FPGA.

	Avail.	Top(12)	*PE*	*PQ*	*Ctrl*	*Exe*
LUTs(K)	859	327	15	9	0.6	5
FFs(K)	1790	412	11	7	0.3	3.7
BRAM	1680	502	117	27	5	0
URAM	800	624	36	36	0	16

VI. Conclusion

In this paper, we present the first FPGA-accelerated maze routing kernel. To leverage the internal parallelism of the algorithm, we design an efficient data structure, hardware-optimized maze routing algorithm, and a scalable hardware-based task scheduler to solve the batched maze routing problem on FPGA. Compared with the state-of-the-art detailed router Dr.CU 2.0 [6], we can accelerate the batched maze routing kernel by up to 3.1×. In the future, we plan to integrate the accelerator to the CPU-FPGA system and leverage system-level parallelism to improve the performance of the entire detailed routing.

Acknowledgements

We sincerely thank Assoc. Prof. Jun Lin in Nanjing University for his helpful feedback. This work is supported in part by the National Science Foundation of China (62034007, 62004006, 61774082), Fundamental Research Funds for the Central Universities (2021300341), and Key Research Plan of Jiangsu Province (BE2019003-4).

References

[1] S. Mantik, G. Posser, W.-K. Chow, Y. Ding, and W.-H. Liu, "Ispd 2018 initial detailed routing contest and benchmarks," in *Proceedings of the 2018 International Symposium on Physical Design*, ser. ISPD '18, 2018, p. 140–143.

[2] H.-Y. Chen and Y.-W. Chang, "Global and detailed routing," in *Electronic Design Automation*. Elsevier, 2009, pp. 687–749.

[3] P.-Y. Wu, W.-K. Mak, T.-C. Wang, C. Zhuo, K. Unda, and Y. Shi, "A routing framework for technology migration with bump encroachment," *Integration*, vol. 58, pp. 1–8, 2017.

[4] G. Chen, C.-W. Pui, H. Li, and E. F. Young, "Dr. cu: Detailed routing by sparse grid graph and minimum-area-captured path search," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[5] A. B. Kahng, L. Wang, and B. Xu, "Tritonroute: An initial detailed router for advanced vlsi technologies," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18, 2018.

[6] H. Li, G. Chen, B. Jiang, J. Chen, and E. F. Y. Young, "Dr. cu 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–7.

[7] Y. Han, K. Chakraborty, and S. Roy, "A global router on gpu architecture," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 78–84.

[8] Y. Han, D. M. Ancajas, K. Chakraborty, and S. Roy, "Exploring high-throughput computing paradigm for global routing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 1, pp. 155–167, 2014.

[9] Y.-J. Chang, Y.-T. Lee, J.-R. Gao, P.-C. Wu, and T.-C. Wang, "Nthu-route 2.0: A robust global router for modern designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 12, pp. 1931–1944, 2010.

[10] D. Korolija and M. Stojilović, "Fpga-assisted deterministic routing for fpgas," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 155–162.

[11] https://aws.amazon.com/ec2/instance-types/f1/ .

[12] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proceedings IEEE INFOCOM 2000.*, vol. 2, 2000, pp. 538–547 vol.2.