2017 IEEE 35th International Conference on Computer Design

Dependency-Aware Parallel Routing for Large-Scale FPGAs

Minghua Shen[1], Nong Xiao[1], and Guojie Luo[2]
School of Data and Computer Science, Sun Yat-sen University, China[1]
Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China[2]
shenmh6@mail.sysu.edu.cn and gluo@pku.edu.cn

***Abstract*—Quantitative effects of Moore's Law have driven qualitative changes in FPGA architecture, applications, and tools. As a consequence, the existing EDA tools takes several hours or even days to implement the applications onto FPGAs. Typically, routing is a very time-consuming process in the EDA design flow. While several attempts have accelerated this process through parallelization, they still do not provide a strong parallel scheme for FPGA routing.**

In this paper we introduce a dependency-aware parallel approach, named Bamboo, to accelerate the routing time for FPGAs. With the dependency detection, Bamboo partitions the nets into multiple subsets, where the nets in the same subsets are independent, and the dependency only exists among different subsets. Specifically, the independent nets in the same subset are routed in parallel, and the subsets are processed in serial according to the original routing ordering. The partitioning problem is solved optimally using dynamic programming, and the parallelization is implemented by speculative parallelism on a single GPU. Experimental results show that our approach achieves an average of 15.13× speedup with negligible influence on the routing quality. Most importantly, it effectively maintains deterministic results and always produces the same results as the serial version.

I. Introduction

FPGA is a popular design style that provides reconfigurable flexibility, lower manufacturing cost and time, and the ability to leverage the advantages of semiconductor process in deep submicron technology. Moreover, FPGA is also a promising hardware accelerator that offers attractive performance and energy for emerging applications that originally ran on general purpose processors. For example, Microsoft has adopted the FPGAs to accelerate the search engine [1]. Central to an FPGA design is EDA tool, which takes the RTL design to implement it into a target FPGA.

FPGA is a pre-fabricated device, and it is cost-effective to maximize the resource usage on the underlying FPGA. Thus the size of RTL design is generally correlated to FPGA size and low utilization is very seldom seen. Typically, the EDA tool has very high utilization stress for mapping the RTL design into target FPGA and meeting the constraints liking timing closure. This trend is to scale in area and performance with Moore's law for each technology node, and today, the largest FPGA can target designs with near to five million logic elements. As a consequence, the existing EDA tools takes several hours or even days to synthesize the RTL design into the target FPGA device.

Routing is probably the most complex and time-consuming process in the FPGA EDA flow. Since routing quality directly affects the maximum clock frequency and other design metrics such as routability and power, it also becomes a critical step in the design cycle. Finding a legal routing solution is equivalent to the NP-complete disjoint path problem in graph theory. The variant of the negotiation-based PathFinder routing algorithm [2] is in dominant use in the commercial FPGA EDA tools. This algorithm enables the different nets to negotiate with each other to find a feasible solution. It invokes a maze expansion step [3] to find a routing tree to connect the individual pins of a single net on the routing resource graph. Specifically, the PathFinder routing algorithm is sequential in nature and lengthy in runtime.

Parallelization is a promising direction to accelerate the routing time but it is non-trivial due to there is a dependency between two or more nets. In the parallel routing process of different nets, the dependency state must be synchronized among threads or processes to avoid sharing the same routing resources to find a feasible solution. Two primary methods have been explored: coarse-grained distributed-memory and fine-grained shared-memory parallelization [4]. The former uses the multi-process techniques to parallelize the multiple nets routing concurrently, but the synchronization overhead is costly due to the high remote memory access latency. The latter leverages the multi-thread techniques to accelerate the single net routing serially. While the achievable speedup may be restricted by the number of processor cores, we explore parallel routing on many-core GPU platform.

In this paper we focus on the fine-grained shared-memory parallel model and introduce a dependency-aware parallel approach, named Bamboo, to accelerate the routing time for FPGAs. Bamboo leverages dependency detection to partition the independent nets that are combined into a single net, which can be speculatively parallelized on GPU. Notably, the partitioning enables the Bamboo is deterministic that always returns exactly the same result when running on the different processors with various cores. Also, it guarantees the Bamboo is serial equivalency that always gives the same answer as the serial version of the algorithm, regardless of how many processing cores are used. They are both important metrics for easier regression verification and customer support in FPGA development. Following are the major contributions of our work to parallel FPGA routing research:

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.45

- We provide an efficient detection approach to determine the net dependency, and impose the strictly-ordered and independent constraints to develop an optimal partitioning algorithm.
- We propose the dependency-aware parallel approach to enable efficient routing acceleration for FPGAs. This parallel router is deterministic and serial equivalency.
- We demonstrate the promising acceleration in routing time for a set of large-scale benchmarks. Experimental results shows that our approach provides an average speedup of 15.13× using speculative parallelism with a single GPU.

II. Motivation and Background

In this section, we give the motivation and background to design a strong parallel routing scheme for FPGAs.

A. Motivation

With many-core processors become increasingly prevalent, parallel computing is applied in FPGA EDA algorithms to improve the performance. Parallelizing FPGA routing involves the following important requirements which are not fully encountered in prior parallel routing works.

1. **Quality.** Most FPGA designers are not tolerate the significant degradation in routing quality compared to serial algorithm. The serial routing algorithm currently optimize wirelength, critical path delay, and any replacement need to deliver equivalent quality. Creating a new parallel router from scratch with equal capability and quality is non-trivial. Existing works [12], [13] result in unacceptable degradation of the routing quality.
2. **Scalability.** Due to the FPGA logic capacity steadily increase at the regular rate, scalability has become an important metric in physical design. Some works [7], [9] attempts to partition the routing resources into the disjoint subsets, where each net is allocated one of the subsets, and routing is performed using only the resources in the subset. However, some high-fanout nets require the routing resources from multiple subsets and can not be routed in parallel. This behavior might further worsen the parallelism with the increasing number of partitions. Therefore, they are not highly scalable.
3. **Determinism.** The parallel router must be deterministic that always returns exactly the repeatable result when run multiple times on different machines. This requirement is rarely fully implemented in prior works, but is vital to ease program development and debugging in commercial context. There exists a parallel router [8] that guarantees deterministic results only on identical processor, in a sense that the routing results are different when running on different commodity hardware. The majority of parallel routers [11], [12], [13] prefers to accelerate the routing time and indirectly overlooks this characteristic.
4. **Serial equivalency.** It is an even stronger requirement, known as serial equivalency, we can apply to our parallel router. This is the property that the parallel router must give exactly the same answer as the serial router, regardless of how many processing cores are used in different machines. A serial-equivalent parallel router is clearly deterministic as well. Serial equivalency has advantages including easier regression testing and customer support. However, the support of this property was limited or ignored in prior works due to it was considered expensive.

Thus, it is a meaningful and challenging work to design a novel parallel router which satisfies these requirements.

B. FPGA Routing

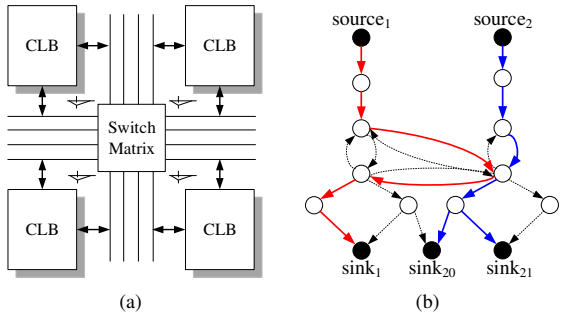

Fig. 1. (a) Architecture; (b) Routing resource graph.

The physical routing resources of an FPGA can be modeled as a directed graph, named routing resource graph, where each vertex represents an electrical pin or a wire segment, and each edge corresponds to a programmable connection between two vertices, as an electrical pin and a wire segment, or a programmable routing switch between two wire segments. Fig. 1(a) shows an example of a small FPGA architecture fragment, and the corresponding routing resource graph is shown in Fig. 1(b) with a channel width of four. The routing is typically performed on the routing resource graph.

Routing is to find disjoint paths in the graph to connect the pins of the source and the sinks for each net. In general, the result of routing a net can be captured by routing resource tree. In FPGA routing, the routing resource tree is a spanning tree of the routing resource graph, which includes all vertices in a routing net. Its root is the source of net, and the sinks are the terminal nodes. Fig. 1(b) shows a net has one source node and a few sinks that are logically connected to the source. The routing resource trees for different nets are disjoint in the graph, to prevent short circuits.

C. PathFinder Algorithm

The negotiation-based PathFinder routing algorithm [2] commonly exists in commercial and academic FPGA EDA research. PathFinder routes one net at a time in each iteration, where dependencies are temporally allowed in the intermediate routing solutions. The nets must negotiate with each other to decide who will make a detour around the dependent resource nodes in subsequent iterations, until all the dependencies are resolved to obtain a complete legal routing solution.

Each iteration rips up an existing routing tree and reroutes it by invoking the maze expansion [3], which computes a path from the source to each sink in the routing resource graph. All of the unvisited vertices are first stored in a priority queue based on their cost, and the vertex with the minimum cost is extracted during maze expansion. If the vertex is a sink, a routing path will be constructed by invoking a backtrace procedure. Otherwise, each neighbor of the vertex, which has not been previously visited, is inserted into the priority queue and the maze expansion continues until a legal routing tree is found.

D. Speculative Parallelism

The speculative parallelism [16] is a thread-level framework that automatically parallelizes a program where the compute and check operators are defined. The general idea of the speculative parallelism is to iteratively apply a set of operators on a subset of elements in the data structure which are referred to as active nodes. At each iteration, the active nodes are performed useful computations, and the rest inactive nodes are idle. The check operator determines whether or not the element assigned to the thread is an active node or not. The compute operator performs the actual computations required for the algorithm to progress and can generate more work by activating inactive nodes. Execution completes when all nodes are inactive and will not be activated again.

Speculative parallelism is an emerging parallel style that uncovers abundant parallelism in irregular computations and simplifies parallel programming. Moctar and Brisk [11] are first to use speculative parallelism with multi-thread techniques to accelerate FPGA routing. Recently, Shen [14] and Hoo [15] have also demonstrated the effectiveness of the speculative parallelism with GPU and MPI techniques for FPGA routing, respectively. Specifically, GPU is excellent at exploiting massive parallelism to achieve high speedup. In this paper we leverage speculative parallelism with GPU techniques to accelerate the routing for FPGAs.

III. Bamboo Approach

With the dependency detection, Bamboo partitions the nets into small subsets and the independent nets of subsets are combined into a single net. An iterative routine is applied to converge a congestion-free state when these nets are routed in parallel. The routine integrates the commonly used negotiation-based ripped-up and re-routed scheme to progressively find the feasible solution. In this section we describe the Bamboo techniques in detail and mainly focus on the dependency-aware parallel routing.

A. Overview

The overall design flow of Bamboo is illustrated in Fig. 2. Bamboo takes the initial netlist as the input, and first checks the dependency of nets based on routing window expansion. Bamboo then separates the nets into a number of subsets using strictly-ordered partitioning. Since the subsets have a number of independent nets, Bamboo combines the independent nets into a single net, which can be routed using speculative parallelism on GPU. Bamboo iteratively performs these four steps until to find a feasible solution.

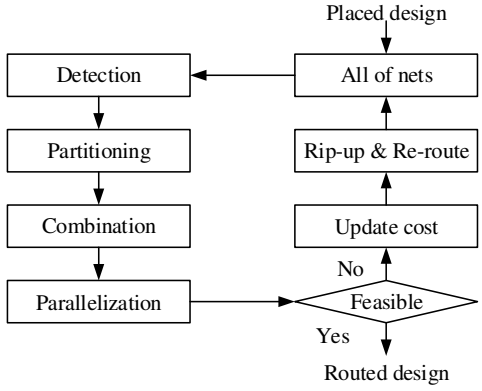

Fig. 2. The parallel routing flow of Bamboo.

Specifically, the partitioning is subject to the original net ordering, the nets in the same subset are independent, and the dependent nets are distributed in different subsets to guarantee the same routing results. In the parallelization, the nets in the same subset are routed in parallel, and the subsets are routed in serial. With the partitioning and parallelization, Bamboo effectively accelerates the routing and satisfies the requirements detailed in Section II-A.

When the iteration proceeds, the dependencies among nets may change. In every iteration, Bamboo repartitions the nets into subsets considering such change to improve the parallelism. All the routing resources are accessible by every net in a shared-memory model during parallelization.

B. Routing Window Detection

The objective of detection is to determine the dependency of nets such that the independent nets are partitioned for parallelization. It is non-trivial to detect whether the nets are independent due to the dependency varies continuously with routing iteration proceeds. One possible way is to run the routing and analyze the results. Although it is correct, it is not practicable because the dependency should be obtained before routing to guide the partitioning. Here we use the routing window expansion techniques to simulate the variation of net dependency.

To enable an efficient detection, we impose some artificial constraints during parallel routing. For each net r_i, we assume a routing window b_i, in which we artificially restrict the exploration of its routing tree. For each routing window b_i, the width and height are w_i^b and h_i^b, respectively, and the lower-left cornet position is at (x_i^b,y_i^b).

Definition 1. (Independent). The nets are independent, if the routing window of every pair of nets do not overlap with each other, i.e.,

$$(x_i^b + w_i^b \leqslant x_j^b) \vee (y_i^b + h_i^b \leqslant y_j^b) \vee$$
$$(x_j^b + w_j^b \leqslant x_i^b) \vee (y_j^b + h_j^b \leqslant y_i^b) \vee$$

It is evident that if the nets satisfies the independent property, it is safe to route them in parallel without affecting the routing results. Thus we resort to whether the routing windows are overlap to quickly determine whether the nets are independent. With the routing iteration proceeds, the routing window is from the initial bounding box of the pins expand to the final bounding box of the routing resource tree. It also means that the nets need to be repartitioned in each iteration to preserve the accurate independency such that Bamboo achieves further speedup.

A simple simulation of routing window expansion is to continuously expand the routing window during iterations. However, this implementation will result in more dependencies between nets due to some unnecessarily larger routing windows and increase the excessive routing time. To solve the issues, the edge recognition is used in conjunction with routing window expansion to decide whether the current routing window continues to be expanded.

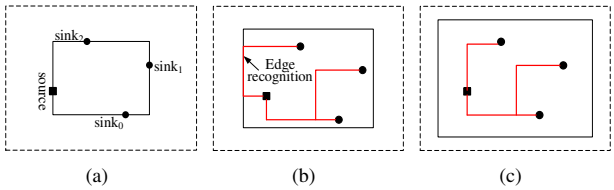

Fig. 3. (a) Initial routing window; (b) Intermediate routing window with edge recognition; (c) Final routing window.

Fig. 3 shows the process of routing window expansion, and a net is probably to use more routing resource nodes when its routing resource tree occupies a node on the edge of the current routing window. Once recognition, the routing window will be linear relaxed on four sides by a fixed distance[1] in the next iteration, until it finds the feasible results. With the edge recognition in the routing window expansion, the routing is simulated to converge using the similar number of iterations as the original routing algorithm.

Thus in each iteration we leverage the routing window to detect the dependency of nets such that the independent nets are partitioned for parallelization.

C. Strictly-Ordered Partitioning

Partitioning is a very important step in the parallel routing flow. In this section we give an effective partitioning algorithm to guarantee these requirements presented in Section II-A are satisfied in Bamboo.

In serial routing, the net ordering has an important influence on the quality of results [5]. Moreover, the perturbation of the net ordering may result in divergence as well [6]. To ensure the convergence of the iterative routing and avoid the degradation in the routing quality, we must maintain the original net ordering, and at the meanwhile, only the independent nets are partitioned for parallelization. Thereby, we give the definition:

[1]This distance is empirically set to one, which is sufficient to find a legal routing solution according to the experiments.

DEFINITION 2. (NET ORDERING). The nets r_i are critical elements in parallel routing. The set $N = \{r_1, r_2, \ldots, r_n\}$ is the collection of all the nets, where the nets are partitioned by the increasing ordering as $r_1, r_2, \ldots, r_n$.

Note that the increasing ordering is the original net ordering of serial VPR routing by default. Based on this basic condition, we explore the partitioning method to guarantee that the parallelization phase will not affect the routing results.

The deterministic behavior of a parallel program is completely defined by its serial equivalency [17]. To maintain the serial equivalency of the parallel router, we impose the strictly-ordered property in the partitioning algorithm. After partitioning, the subsets are strictly-ordered, and the nets in the same subset are independent.

DEFINITION 3. (STRICTLY-ORDERED). Given the nets $N = \{r_1, r_2, \ldots, r_n\}$ to be routed, by partitioning, we call the subsets $M = \{s_1, s_2, \ldots, s_m\}$ strictly-ordered, if $s_p = \{r_{i_p}, r_{i_p+1}, \ldots, r_{i_{p+1}-1}\}$, where $1 = i_1 < i_2 < \ldots < i_{m+1} = n + 1$.

We regard these subsets strictly-ordered, because for $i < j$, every net in s_i is routed before a net in s_j as in the original net ordering. For the partitioning subsets M, the parallel router routes the nets in s_1 concurrently at the first subset, and then after synchronization, it routes the nets in s_2 at the second subset, and so on. Notice that strictly-ordered is a necessary condition and independent is a sufficient condition for the parallel router with serial equivalency.

Based on the independent and strictly-ordered conditions, the parallel router is serial equivalency, thanks to the routing results are equivalent to the serial routing results, where the nets $r_1, r_2, ..., r_n$ are routed one by one. A serially equivalent algorithm is clearly deterministic as well.

To implement the parallel routing flow shown in Fig. 2, we formally formulate this partitioning problem as follows:

Problem Formulation. Given the nets N = $\{r_1, r_2, ..., r_n\}$ to be routed, our objective is to find a partition M = $\{s_1, s_2, ..., s_m\}$ with strictly-ordered and independent properties to minimize the total routing time.

Due to the number of subsets M is directly involved with the runtime of parallel routing, we minimize the number of subsets to indirectly minimize the total routing time. This problem can be solved by dynamic programming optimally.

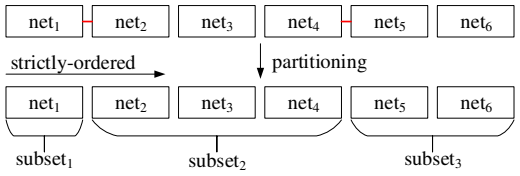

Fig. 4. Strictly-ordered partitioning.

We partition the nets into M subsets s_1, s_2, ..., s_m with r_1, r_2, ..., r_n nets respectively. Note that r_1+r_2+...+r_n = n, and the relative ordering for the subsets must be preserved.

Fig. 4 shows an example that the nets are partitioned into multiple subsets subject to the strictly-ordered and independent properties with each partitioning.

The net partitioning can be solved by dynamic programming in quadratic time. We will analyze the time complexity of the partitioning below. Table I gives the related notations.

TABLE I
NOTATIONS FOR THE PARTITIONING PROBLEM

Notation	Description
$E[j][i]$	The feasibility indicator whether the nets $r_j, r_{j+1}, ..., r_i$ are independent.
$C[i]$	The minimum number of subsets for the nets from r_1 to r_i with the strictly-ordered and independent properties.

Specifically,

$$E[j][i] = \begin{cases} 1, & \text{independent} \\ +\infty, & \text{otherwise} \end{cases}$$

The algorithm consists of two stages: the precomputation stage and the dynamic programming stage. We precompute the $E[j][i]$ using simple pair-wise testing in worst-case quadratic time. In practice, even this simple algorithm terminates very fast, because the size of independent subsets is limited.

Based on the precomputed $E[j][i]$, we can start the dynamic programming algorithm. The minimal number of strictly-ordered subsets of the first i nets satisfies

$$C[i] = \begin{cases} 1, & i = 0 \\ \min_{j=0}^{i-1}\{C[j] + E[j+1][i]\}, & i \geq 1 \end{cases}$$

The solution to our problem is $C[N]$. It is obvious that computing $C[N]$ takes $O(N^2)$, given $E[i][j]$. Thus, the time complexity of the overall dynamic programming algorithm is quadratic.

Correctness. This dynamic programming algorithm enables the partitioning is strictly-ordered. It also can be proved by induction that the strictly-ordered partitioning generates the minimal number of subsets.

Effectiveness. In practice, this partitioning algorithm is fast, because the value of j that needs to be enumerated is very small compared to the nets i. Moreover, due to the sparseness of routing resource graph, there will be much fewer subsets than the independent nets routed in parallel.

With this partitioning, the subsets are *strictly-ordered* to maintain the routing quality, combined with the nets in the same subset are *independent* to guarantee the serial equivalency. The serial equivalent parallel algorithm is deterministic by default. Moreover, our parallel router is clearly scalability based on strictly-ordered partitioning.

D. Combination and Parallelization

With the process of partitioning, we combine the multiple independent nets into a single pseudo net and implement the net routing using speculative parallelism on GPU.

An effective implementation of combination is to introduce a *pseudo* source node to directly connect to the actual source nodes of multiple independent nets. Thus the independent nets of each subset can be combined into a single pseudo net, which can be routed as the real single net by speculative parallelism on GPU. Here we demonstrate the source of speedup when routing multiple nets in parallel. To parallelize a single net, the wait time between beginning and ending phases is costly. When we parallel route multiple nets as a single net, the overhead in these two phrases are reduced to improve the speedup. Fig. 5 shows multiple independent nets are combined for parallelization.

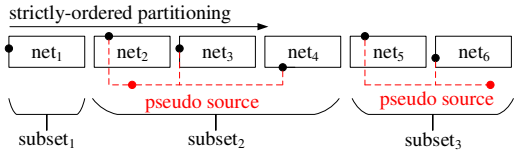

Fig. 5. Combination of multiple independent nets.

With the combination of multiple independent nets based on strictly-ordered partitioning, We collect the independent nets according to their original ordering, and make sure that their parallel routing will not affect their routing results, comparing to the sequential routing. By using the routing windows of the nets, this collection process can efficiently and precisely determine the dependency between the nets in each iteration. With the combination, we start to route the first subset of independent nets concurrently and then route the next subset until all the subsets are processed as shown in Fig. 6. It is evident that this subset can be routed in parallel without affecting the routing results.

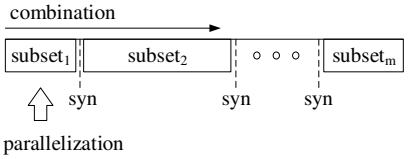

Fig. 6. Parallelization of subsets.

In parallelization, we exploit the speculative parallelism to implement the single net routing on GPU, while the subsets are routed one by one in the original ordering. The single net routing is essentially to solve the shortest path problem. Notice that same results can be obtained when the shortest path solution is unique; when there are multiple paths with the same shortest length, we can always break the tie using the labels of the nodes to make the solution unique.

The details of speculative parallelism implements the single source shortest path on GPU for single net routing are as follows. Instead of the priority queue, the centralized worklist, a variant of the queue, with atomic memory operation (AMO) is used to manage the speculative parallelism.

Fig. 7 shows an example to route a single net using speculative parallelism. An initialization step is first to pre-check all

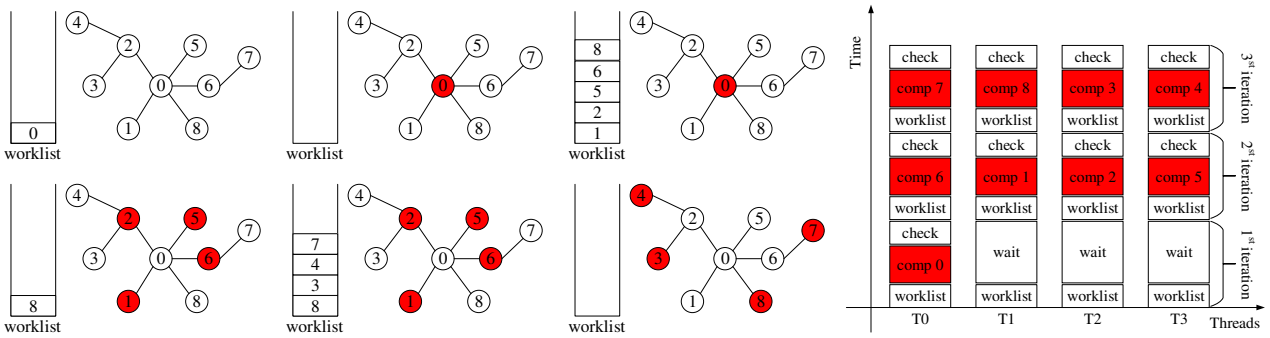

Fig. 7. Route a single net using speculative parallelism.

the nodes and populates the active nodes into the worklist for parallel processing. For example, to route a net, the worklist is initialized with the source node. Second, every thread pulls some active nodes from the worklist using AMO and then applies the compute operator to the corresponding active nodes. For example, in the routing algorithm, the threads pull several active nodes from the worklist and apply the compute operator to update the known shortest path of their neighbors. The newly activated nodes, whose known shortest paths are updated, are pushed onto the worklist with AMOs, such that only active nodes will be visited in the next iteration. Note that in general, the neighborhood of an active node is distinct from its neighbors in the graph. This process is repeated until the worklist becomes empty. And at that meanwhile, the solution of single net routing is obtained.

With the dependent detection of routing window, the nets are partitioned into subsets. Specifically, the independent nets in subsets are combined into a single net, which can be routed using speculative parallelism on GPU. With the requirement of independent and strictly-ordered, this parallel router is aware of net dependency and produces serial equivalent results.

IV. Evaluation

In this section we evaluate and analyze the experimental results of Bamboo, a novel dependency-aware parallel router for FPGAs.

A. Experimental Setup

The experiments are performed on the Linux server with a 6-core Intel Xeon E5-2620 CPU at 2.2GHz and 32 GB shared memory, equipped with a Tesla K40c GPU having 2880 cores in 15 streaming multiprocessors and 12 GB video memory. The parallel routing approach described in this paper is implemented with C++ and CUDA. Some of the GPU implementations of the SSSP algorithm are adapted from the source code in the LonestarGPU collection [18]. Table II shows a summary of the 11 largest representative benchmarks used for the experiments. Specifically, these circuits are from the large-scale VTR benchmark suite [19] commonly used in FPGA EDA research. The MCNC benchmarks [20] are not evaluated because even the largest case takes several minutes to route. We use ABC for logic synthesis and technology mapping, T-VPack for packing, and VPR placer for placement, respectively.

TABLE II
Benchmark Summary

Bench.	Arch.	Dim.	Nets	CLBs
mkDelayW.	k4_N4_90nm	48x48	5224	1554
blob_mer.	k4_N4_90nm	51x51	6606	2702
mkSMAdap.	k4_N4_90nm	53x53	7154	3126
mkPKtMer.	k4_N4_90nm	58x58	7474	3767
or1200	k4_N4_90nm	65x65	8078	3648
stereov.0	k6_N10_40nm	39x39	9312	1492
stereov.1	k6_N10_40nm	39x39	13523	1401
LU8PEEng	k6_N10_40nm	53x53	16278	2373
bgm	k6_N10_40nm	73x73	27853	4225
stereov.2	k6_N10_40nm	86x86	36479	2802
mcml	k6_N10_40nm	101x101	81282	7934

The baseline for comparison is the original VPR 7.0 router [19], which is faster than the existing commercial routers such as Quartus router [21]. Across all runs, each benchmark is routed using a channel width of $1.4\times$ the minimum channel width needed by VPR, following the same configurations as in the previous works [11], [12]. Although the focus of this paper is parallelizing the FPGA routing, since the dependency-aware parallel router is with strictly-ordered and independent constraints, it can work and maintains the same routing results as serial router.

B. Experimental Results

Fig. 8 reports the speedups provided by Bamboo, with VPR router as the baseline. Bamboo integrates detection, partitioning, combination and parallelization techniques to accelerate the routing with the constraints of strictly-ordered and independency. It can be seen that Bamboo achieves an average speedup of $15.13\times$ on a single GPU.

To analyze the individual influence of these techniques, we implement the serial and parallel versions, respectively. The baseline, original VPR router, is executed on overall routing resource graph. But our serial version runs on the routing window. The window constraints the routing search space to optimize the runtime. On average, the serial version provides a $1.53\times$ speedup. Without partitioning and combination, the parallel version[2] implements the routing in net-by-net fashion,

[2]For comparison with Bamboo, the parallel version uses the routing window expansion techniques as well.

TABLE III
ROUTING QUALITY SUMMARY

Quality	Wirelength				Critical Path Delay(ns)			
Bench.	Baseline	Serial	Parallel	Bamboo	Baseline	Serial	Parallel	Bamboo
mkDelayW.	112122	112931	112931	112931	8.575	8.663	8.663	8.663
blob_mer.	119927	120836	120836	120836	19.35	19.54	19.54	19.54
mkSMAdap.	108533	109237	109237	109237	19.46	19.65	19.65	19.65
mkPKtMer.	109980	110791	110791	110791	14.93	15.09	15.09	15.09
or1200	133856	134665	134665	134665	66.99	67.65	67.65	67.65
stereov.0	115870	116582	116582	116582	4.745	4.795	4.795	4.795
stereov.1	199814	201525	201525	201525	6.795	6.871	6.871	6.871
LU8PEEng	426520	429667	429667	429667	124.1	125.4	125.4	125.4
bgm	152096	153405	153405	153405	33.57	33.97	33.97	33.97
stereov.2	702836	708573	708573	708573	16.60	16.78	16.78	16.78
mcml	1542736	1583886	1583886	1583886	48.82	49.29	49.29	49.29
Avg.	1.0000	1.0073	1.0073	1.0073	1.0000	1.0105	1.0105	1.0105

which achieves average speedup of 7.09× with a single GPU. The Bamboo parallel router can produce a 2.13× extra improvement, although the time complexity of partitioning algorithm is quadratic.

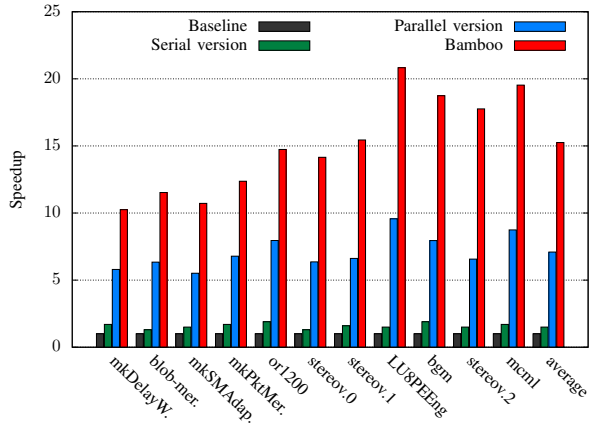

Fig. 8. Speedup of Bamboo.

In Fig. 8, we also observe that the speedup of Bamboo scales well when the design size grows. We do obtain notable speedups for the four largest benchmarks in Bamboo, because more independent nets can be combined into a single pseudo net and result in more acceleration on GPU. These results indicate that Bamboo is promising to maintain a similar speedup on very-large-scale FPGA design.

Table. III reports the quality of Bamboo compared to the baseline VPR router, regarding the routed wirelength and the critical path delay. Only a minor difference of routed wirelength between the baseline and Bamboo is observed for all benchmarks. For the critical path delay, it is worsened about 1% on average. Though Bamboo generates serial equivalent routing results, its serial version also uses the technique of routing window expansion. The window constraints the search space and change the routing path, thus, the routing results are different from original VPR router. From the results of the routed wirelength and the critical path delay, we conclude that Bamboo has negligible impact on the routing quality.

The previous work [22] reports that how much quality degradation is necessary to achieve runtime improvement for FPGA routing. It is acceptable that Bamboo trades the 1% degradation in the critical path delay for the 15.13× speedup. Furthermore, Xilinx recent announces that the routing resources will increase to 2× and more for FPGAs.

C. Comparison with Existing Works

Table. IV shows the comparisons of Bamboo with recent parallel routing works in quality, scalability, determinism, serial equivalency, and speedup. Bamboo maintains the serial equivalency and thus gives exactly the same answer as the corresponding serial router. Notice that the existing works guarantees the deterministic results only using the identical processor, and thus they are weak deterministic. Bamboo is strong deterministic, regardless of how many processing cores are used in different processors.

TABLE IV
COMPARISON WITH EXISTING WORKS

Comparison	Qual.	Scal.	Deter.	Seri. Equi.	Speedup
FPT'10 [8]	-1%	Yes	Weak	No	2×
FPL'13 [10]	-2%	No	Weak	No	2×
DAC'14 [11]	-1%	Yes	No	No	5×
ICCAD'15 [12]	-10%	Yes	Weak	No	7×
FPL'15 [13]	-8%	Yes	Weak	No	7×
FPGA'17 [14]	-2%	Yes	Weak	No	18×
FCCM'17 [15]	-1%	Yes	Weak	No	19×
Bamboo	-1%	Yes	Strong	Yes	15×

Finally, it can be seen that our parallel router is close to the state-of-the-art techniques in terms of speedup.

V. RELATED WORK

There are a few other academic works that have accelerated FPGA routing through parallelization. These parallel routers are mainly based on coarse-grained distributed-memory model and fine-grained shared-memory model.

In distributed-memory parallel routing, Chan and Schlag [6] are the first to accelerate the routing. Although the results are not deterministic, they achieve impressive speedup results of 2.5× using three processors. And then, Cabral et al [7] describes an efficient parallel routing algorithm specifically for the architecture with disjoint topology. As expected, the

algorithm achieves the almost linear speedup, however, the routability is not good.

Another influential work by Gort and Anderson [8], [9] guarantees the deterministic results on the identical processor. They achieve a 2.8× speedup using eight cores, although their method is not highly scalable. Recently, Shen and Luo [12] leverages a dynamic programming algorithm to determine the optimal recursive partitioning strategy. The parallel router exploits more parallelism and provides an average speedup of 7×, although it degrades the quality of routed wirelength.

The most recent work by Hoo and Kumar [15] presents a distributed-memory parallel router based on speculative parallelism and path encoding. This parallel router achieves an average speedup of 19× with 32 processes, although it can not produce the serial equivalent routing results.

In shared-memory parallel routing, an effort by Zhu et al [10] partitions the high-fanout nets into several low-fanout subnets to be routed in parallel. They achieve a speedup of 1.9× on a quad-core processor platform with 2.3% loss in solution quality. And then, Moctar and Brisk [11] exploits the operator formulation to accelerate the routing for FPGAs. They achieve a good speedup of 5.4× using eight threads and the results is non-deterministic.

Another effective work by Hoo et al [13] proposes a fully parallel router based on Lagrangian relaxation. They attempt to partition the original routing problem into independent subproblems. This approach produces an average speedup of 7× using eight threads, compared to its sequential version. The most recent shared-memory work by Shen and Luo [14] explores GPU-accelerated parallel routing based on subgraph dynamic expansion. This parallel router achieves an average speedup of 18× on a single GPU, although it can not maintain the serial equivalency as well.

In this paper we focus on the fine-grained shared-memory parallel model and design a dependency-aware parallel routing approach for FPGAs. This parallel router guarantees the serial equivalency and achieves an average speedup of 15× on a single GPU. Also, this parallel router is first work to maintain the serial equivalency and strong determinism.

VI. Conclusion and Future Work

Parallelization is a very promising direction to accelerate the routing time for FPGAs. Efficient parallel router provides not only good speedup, but also deterministic and serial equivalent routing results.

In this paper we present a dependency-aware parallel routing approach, named Bamboo, to implement the acceleration and produce the serial equivalent routing results. Bamboo first determines the net dependency based on routing window expansion and then partitions the nets into multiple subsets with requirements of independent and strictly-ordered. The independent nets in the same subsets are combined into a single net, which can be routed by speculative parallelism on GPU. The dependent nets are distributed in the different subsets, which can be processed in serial according to the original routing ordering. Specifically, we propose a provably optimal partitioning algorithm to minimize the number of subsets to minimize the routing time. Experimental results shows that Bamboo provides an average of 15× speedup with a tolerable loss in the routing quality.

In the future our work can be enhanced in two ways, including 1) exploring the techniques to extract independent nets for the more acceleration of multi-net routing, and 2) leveraging similar ideas to explore the parallelization techniques for an FPGA-accelerated FPGA router.

VII. Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive comments. This work is supported by NSFC61520106004, NSFC61433019, 2016YFB1000302, and 20176700041110006.

References

[1] Adrian M. Caulfield et al. “A cloud-scale acceleration architecture.” in International Symposium on Microarchitecture (MICRO), 2016.
[2] L. McMurchie and C. Ebeling. “Pathfinder: A negotiation-based performance-driven router for FPGAs.” in International Symposium on Field-Programmable Gate Arrays (FPGA), Feb. 1995.
[3] C. Lee. “An algorithm for path connections and its applications.” in IRE Trans. on Electronic Computers, 1961.
[4] B. Catanzaro, K. Keutzer, and B. Su. “Parallelizaing CAD: A timely research agenda for EDA.” in Design Automation Conference (DAC), Jun. 2008.
[5] R. Rubin and A. Dehon. *Timing-driven pathfinder pathology and remediation:quantifying and reducing delays noise in VPR-pathfinder.* in International Symposium on Field-Programmable Gate Arrays (FPGA), Feb. 2011.
[6] P. Chan, M. Schlag, C. Ebeling, and L. McMurchie. *Distributed-memory parallel routing for field-programmable gate arrays.* in Trans. on Computer-Aided Design (TCAD), 2000.
[7] L. Cabral, J. Aude, and N. Maculan. “TDR: A distributed-memory parallel routing algorithm for FPGAs.” in International Conference on Field Programmable Logic and Applications (FPL), Sept. 2002.
[8] M. Gort and J. Anderson. “Deterministic multi-core parallel routing for FPGAs.” in International Conference on Field-Programmable Technology (FPT), Dec. 2010.
[9] M. Gort and J. Anderson. “Accelerating FPGA routing through parallelization and engineering enhancements special section on PAR-CAD 2010.” in Trans. on Computer-Aided Design. (TCAD), 2012.
[10] C. Zhu, J. Wang, and J. Lai. *A novel net-partition-based multi-threaded FPGA routing method.* in International Conference on Field Programmable Logic and Applications (FPL), 2013.
[11] Y. Moctar and P. Brisk. “Parallel FPGA routing based on the operator formulation.” in Design Automation Conference (DAC), Jun. 2014.
[12] M. Shen and G. Luo. “Accelerate FPGA routing with parallel recursive partitioning.” in International Conference on Computer-Aided Design (ICCAD), Nov. 2015.
[13] C. Hoo, A. Kumar, and Y. Ha. “ParaLaR: A parallel FPGA router based on lagrangian relaxation.” in International Conference on Field Programmable Logic and Applications (FPL), Sept. 2015.
[14] M. Shen and G. Luo. *Corolla: GPU-accelerated FPGA routing based on subgraph dynamic expansion.* in International Symposium on Field-Programmable Gate Arrays (FPGA), Feb. 2017.
[15] C. Hoo and A. Kumar *ParaDiMe: A Distributed Memory FPGA Router Based on Speculative Parallelism and Path Encoding.* in International Symposium on Field-Programmable Custom Computing Machines (FCCM), Apr. 2017.
[16] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Lojo, D. Prountzos, and X. Sui. “The tao of parallelism in algorithms.” in Conference on Programming Language Design and Implementation (PLDI), 2011.
[17] R. L. Bocchino Jr., V. S. Adve, S. V. Adve, and M. Snir. *Parallel programming must be deterministic by default.* in International Conference on Hot Topics in Parallelism, 2009.
[18] M. Burtscher, R. Nasre, and K. Pingali. *A quantitative study of irregular programs on GPUs.* in International Symposium on Workload Characterization, 2012.
[19] J. Rose et al. *VPR 7.0: Next generation architecture and CAD system for FPGAs.* in Trans. on Reconfigurable Technology and Systems (TRETS), 2014.
[20] S. Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0.* Microelectronics Center for North Carolina (MCNC), 1991.
[21] K. Murray, S. Whitty, S. Liu, J. Luu and V. Betz. *Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD.* in Trans. on Reconfigurable Technology and Systems (TRETS), 2015.
[22] C. Mulpuri and S. Hauck. *Runtime and quality tradeoffs in FPGA placement and routing.* in International Symposium on Field-Programmable Gate Arrays (FPGA), 2001.