Exploring GPU-Accelerated Routing for FPGAs

Minghua Shen, *Member, IEEE*, Guojie Luo, *Member, IEEE*, and Nong Xiao, *Senior Member, IEEE*

Abstract—Field Programmable Gate Arrays (FPGAs) are reconfigurable architectures able to provide a good balance between energy efficiency and flexibility with respect to CPUs and ASICs. The main drawback in using FPGAs, however, is their timing-consuming routing process, significantly hindering the designer productivity. An emerging solution to this problem is to accelerate the routing by parallelization. Existing attempts of parallelizing the FPGA routing either do not fully exploit the parallelism or suffer from an excessive quality loss. Massive parallelism using GPUs has the potential to solve this issue but faces non-trivial challenges. To cope with these challenges, this paper explores GPU-accelerated routing approach for FPGAs. We leverage the idea of problem size reduction by limiting the single-net routing in a small subgraph rather than in an entire graph, further enabling the GPU-friendly shortest path algorithm to be used in FPGA routing. We maintain the convergence after problem size reduction by using the dynamic expansion of the routing resource subgraph, where the routing region of subgraph will be progressively expanded to find a feasible solution to each net. In addition, we are based on a GPU platform to explore the fine-grained single-net parallel routing in three ways and propose a hybrid approach to combine the static and dynamic parallelization for better speedup in FPGA routing. To explore the coarse-grained multi-net parallelization, We propose a dynamic programming-based partitioning algorithm to parallelize the routing of multiple nets while generating the equivalent routing results as the original single-net routing. Experimental results show that our proposed approach can provide an average of about 21.53× speedup on a single GPU with a tolerable loss in the routing quality and maintain a scalable speedup on large-scale routing resource graphs. To our knowledge, this is the first work to demonstrate the effectiveness of GPU-accelerated routing for FPGAs.

Index Terms—Hardware, reconfigurable architectures, FPGAs, routing, GPU parallelization

1 Introduction

With the reaching of the end of Moore's law and Dennard scaling [1], computing landscape is becoming increasingly parallel and heterogeneous, consisting of a larger number of cores and customized accelerators. Field Programmable Gate Arrays (FPGAs) shows particularly promising as an acceleration technology with its reconfigurability, owing to that they can improve the energy efficiency and performance in a broad range of applications [2], [3], [4]. For example, Microsoft's large-scale FPGA-based cluster has been used thus far to accelerate Bing web search engine and deep neural network processing [5], [6]. Compared with other competitive accelerators like GPUs, FPGAs usually offer much better energy efficiency and can still deliver high performance in datacenters. However, the increasingly lengthy compilation time associated with FPGA computer aided design (CAD) algorithms has been a severe limitation to broader adoption of this technology.

- *M. Shen is with the School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510275, China, and is also with the Key Laboratory of Machine Intelligence and Advanced Computing, Ministry of Education, Guangzhou 510275, China. E-mail: shenmh6@mail.sysu.edu.cn.*
- *G. Luo is with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China. E-mail: gluo@pku.edu.cn.*
- *N. Xiao is with the School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510275, China. E-mail: xiaon6@mail.sysu.edu.cn.*

Manuscript received 28 Mar. 2018; revised 22 Oct. 2018; accepted 30 Nov. 2018. Date of publication 7 Dec. 2018; date of current version 15 May 2019. (Corresponding author: Minghua Shen.)
Recommended for acceptance by M. Smith.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2885745

Fig. 1a shows a representative FPGA CAD flow. During logic synthesis and technology mapping, a circuit is translated into a netlist composed of lookup tables (LUTs) and fip-fops (FFs). In the packing stage, several LUTs and FFs together form a basic logic element (BLE) and then several BLEs are grouped into a configurable logic block (CLB). After packing, placement is responsible for determining the physical position of all CLBs in a given FPGA. Finally, routing is to assign wire segments and select programmable switches to construct the required connections among the logic components.

Routing is the most time-consuming stage in the FPGA CAD flow [7]. Fig. 1b shows the average proportion of execution time using state-of-the-art academic VTR tools to compile ten large circuit designs from VTR benchmark suite [43]. Since the final 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. The PathFinder routing algorithm [9] is in dominant use in the FPGA communities due to its superior performance and quality of results. This algorithm enables the nets to negotiate with each other to find a feasible routing solution. However, routing is a very lengthy process in terms of runtime and a promising direction to overcome the runtime challenge is through parallelization [10]. Several recent attempts on parallelizing the FPGA routing have been reported [11], [12], [13], [14], [15], [16]. However, there is a lack of literature on GPU acceleration of FPGA routing. In this paper, we explore how to use GPU efficiently for a very fast FPGA routing approach.

Graphics Processing Units (GPUs) offers a massively parallel computing platform to address the time-consuming

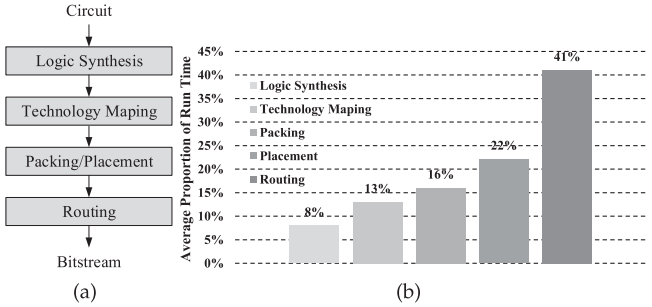

Fig. 1. (a) A representative FPGA CAD flow. (b) Average proportion of run time to each design stage.

and tedious problems [17]. GPU acceleration techniques have shown excellent performance in applications with the data parallel paradigm. Several algorithms in the area of FPGA CAD have been successful accelerated using GPUs [18], [19], [20], [23]. However, the PathFinder routing algorithm for FPGAs is sequential in nature. Dependencies exist in the routing process of different nets, as well as the routing of a single net. Such dependencies violate the requirement of independency in data parallel paradigm and thus, the algorithm must be thoroughly revised to take full advantage of GPU acceleration techniques.

The kernel of FPGA routing is, in fact, a single source shortest path (SSSP) solver. Several GPU-based approaches have been proposed to accelerate the SSSP solver [21], [22], but the available speedup is insignificant due to the synchronization overhead is costly and the memory access is irregular [23]. Also, the GPU-accelerated path-finding solvers for video games [24] and the global routing problem for ASICs [25] assume the routing structures as rectilinear grids. Therefore, these parallelization techniques cannot be directly applied to FPGA routing, whose complex routing resources form a general graph. Among the GPU-based SSSP solvers for general graphs, the Bellman-Ford algorithm provides the greatest speedup so far [26], [27], although its worst-case time complexity is inferior to the Dijkstra's algorithm. Moreover, the serial Bellman-Ford algorithm excels when running on a small graph [28].

In this paper, we leverage multiple techniques to enable the usage of the GPU-friendly Bellman-Ford algorithm to increase the speedup and restrict its weakness. Specifically, we observe that the bounding box of the final routing tree of most nets is only slightly larger than the bounding box of the terminal pins. Thus, we can use the Bellman-Ford algorithm in a small subgraph defined by a limited-size bounding box to replace the original Dijkstra or A* algorithm in FPGA routing. We make use of such observation and idea in our GPU-accelerated routing approach. The dynamic expansion strategy of the routing resource subgraph is applied to control the problem size so that we can adopt GPU-based Bellman-Ford algorithm to achieve a better speedup to route a single net. Moreover, we explore the fine-grained node and edge parallelism in the routing of a single net, and we discuss the possibility to leverage coarse-grained net parallelism to route multiple nets concurrently.

In summary, we explore the novel GPU acceleration techniques for FPGA routing. The main contributions of this work are described as follows:

- The GPU-friendly Bellman-Ford algorithm becomes practical for FPGA routing, attributed to our problem size reduction technique by exploiting the coverage estimation and dynamic expansion on the routing resource subgraphs.
- We further improve the speedup by considering the single-net and multi-net parallelism. On single-net parallelization, we propose a hybrid approach that combines the advantages of both the static and dynamic parallelism in the SSSP solver for FPGA routing. On multi-net parallelization, we present a deterministic net-parallel technique that guarantees the equivalent routing results as the original ordered net-by-net routing.
- The proposed method provides an average of 21.53× speedup on GPU. We also analyze it scalability using large-scale routing graphs. To our knowledge, this is the first work on utilizing GPU to accelerate the routing time for FPGAs.

The preliminary version has been presented at International Symposium on Field Programmable Gate Arrays (FPGA) in 2017 [29]. Notice that in this paper, we not only present novel insights into the GPU-friendly routing, but also propose an optimal dynamic programming-based partitioning algorithm to explore multi-net parallelism for fast FPGA routing. Evaluations have shown that the achieved speedup is further improved significantly. Relying on GPU acceleration, we achieves significantly greater speedups than the publicly available VPR 7.0 router [43] and the state-of-the-art VPR-based parallel routers. We also believe that it will have many useful applications for fast compilations due to the fundamental importance of routing.

The rest of the paper is organized as follows: Section 2 gives the background and motivation; Section 3 presents the methodologies of subgraph dynamic expansion; Section 4 explores the GPU-accelerated routing techniques; Section 5.2 discusses the experimental results; Section 6 shows the related work on parallel routing and Section 7 concludes the paper.

2 Background and Motivation

2.1 Routing Problem

The physical routing resources of an FPGA can be modeled as a directed graph $G(V, E)$, named *routing resource graph*, where each vertex v_i represents an electrical pin or a wire segment and each edge e_{ij} corresponds to a programmable connection between an electrical pin and a wire segment, or a programmable routing switch between two wire segments. Fig. 2a shows an example of a partial routing architecture and its channel width is set to two, and the routing resource graph is shown in Fig. 2b.

The routing problem is to find disjoint paths in $G(V, E)$ to connect the pins of the source and the sinks for each net in Fig. 2b. A net N_i has one source node s_i and a few sinks t_{ij} that are logically connected to the source. Both the source and sinks are vertices in V, and thus, the net N_i is a subset of V. The routing of net N_i is to find a subtree in graph G that includes all vertices in N_i, and this subtree is called the routing tree RT_i of net N_i. The source s_i is the root node of RT_i, and the sinks t_{ij} are the terminal nodes. The routing

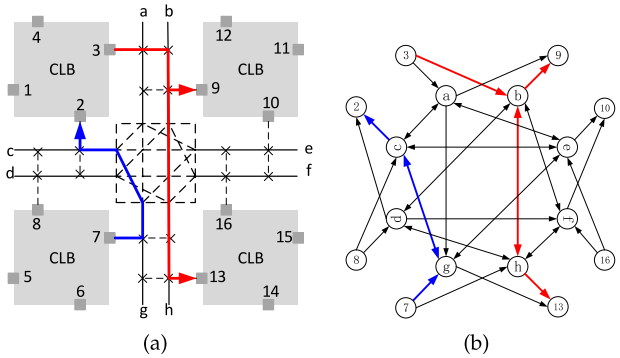

Fig. 2. FPGA routing resource graph. (a) Architecture. (b) Partial routing graph.

trees for different nets are disjoint in G, to prevent short circuits. This routing problem is NP-complete [14].

2.2 PathFinder Algorithm

A summary is described below on the negotiation-based PathFinder routing algorithm [9]. PathFinder routes one net at a time in each iteration, where congestions are temporally allowed in the intermediate routing solutions. The nets must negotiate with each other to decide who will make a detour around the congested resource nodes in subsequent iterations, until all the congestions 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 [8], which computes a path from the source to each sink in the routing resource graph. It is also the most expensive task in FPGA routing. All of the unvisited vertices are first stored in a priority queue based on their cost, and the vertex v_{min} with the minimum cost is extracted during maze expansion. If v_{min} is a sink, a routing path will be constructed by invoking a backtrace procedure. Otherwise, each neighbor v of v_{min}, which has not been previously visited, is inserted into the priority queue and the maze expansion continues until a legal routing tree is found.

The PathFinder routing, a tedious and time-consuming process, is inherently sequential and, since it operates on graphs, it is irregular. These make it very challenging to parallelize the net routing. In coarse grain, the congestion costs are sequentially updated net after net within a routing iteration. While in fine grain, the priority queue in the maze expansion routing of a single net limits the practical concurrency. Such data sharing in the coarse grain and the fine grain violates the requirement of data independency of the GPU-friendly data-parallel paradigm. Thus, the existing routing algorithms are not designed for GPU acceleration and must be revisited.

2.3 Dynamic Parallelism

Dynamic parallelism is an important and useful technique able to dynamically exploit the parallelism in irregular computations such as graph algorithms. Moctar and Brisk [14] have demonstrated the effectiveness of dynamic parallelism with multi-threading techniques by using the operator formulation [32] for FPGA routing.

The general idea of operator formulation to implement dynamic parallelism is to apply a *compute* operator

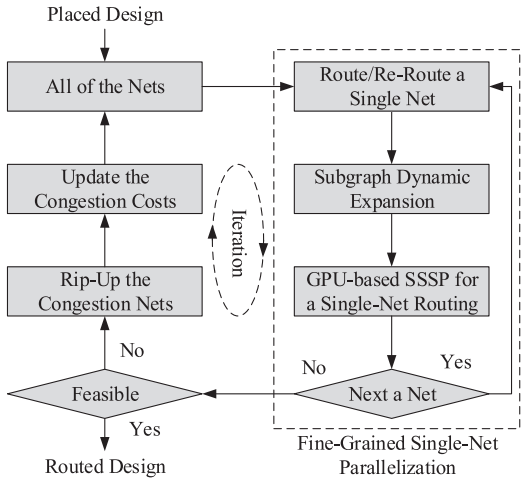

Fig. 3. The overall design flow of GPU-accelerated routing for FPGAs.

iteratively on a subset of nodes in a graph. At each iteration the active nodes perform useful computations and the rest inactive nodes are idle. A *check* operator determines whether a node is active or inactive. The compute operator often accesses neighboring nodes and can activate inactive nodes for further processing. Execution completes when all nodes are inactive and will not be activated again.

For example, in the Bellman-Ford algorithm, a compute operator updates the known shortest path of a node, and a check operator checks whether an upstream of a given node has an updated known shortest path. The operator formulation is a framework that automatically parallelizes a program where the compute and check operators are defined.

In parallel routing exploration, the dynamic parallelism of routing resource nodes and edges will be inspired to accelerate GPU-based FPGA routing.

2.4 Overall Design Flow

The overall design flow of the proposed approach is shown in Fig. 3. Note that we still preserve the negotiation-based framework [9] that iteratively reduces the routing resource congestions by ripping-up and re-routing the nets until to find a feasible routing solution. In subgraph dynamic expansion, the routing subgraph of each net is extracted according to the *initial coverage* strategy at the first iteration and its size may be expanded according to the *dynamic expansion* strategy. In the GPU-based SSSP for a single net routing, we propose multiple techniques to obtain a high degree of parallelism and significant speedup. We explore the node and edge parallelism and a hybrid approach is proposed to accelerate the single-net routing on GPU. We also leverage the net parallelism to accelerate the multi-net routing. The kernel of the proposed approach is the GPU-friendly SSSP algorithm to route every net inside its own routing subgraph region.

In this paper, we explore the capability of GPU acceleration for FPGA routing. The computational kernel in FPGA routing is the single-source shortest path (SSSP) solver. First, we present the subgraph dynamic expansion method to enable the use of a GPU-friendly SSSP algorithm for FPGA routing in Section 3. Second, we explore different GPU-based parallelizations and propose an efficient hybrid solution, followed by multi-net parallelization in Section 4.

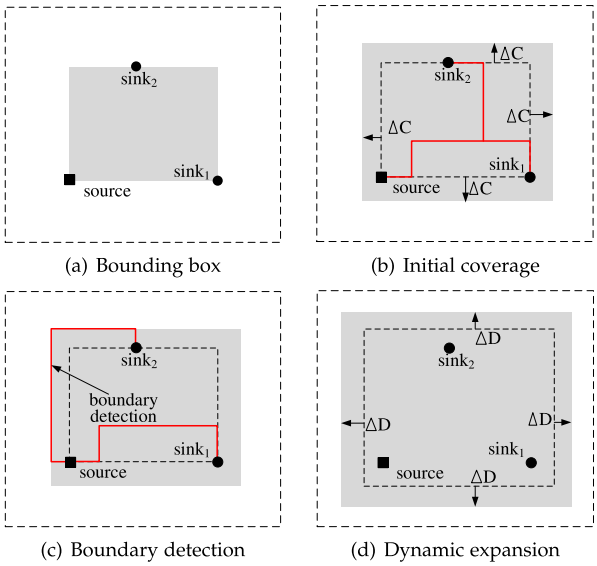

Fig. 4. The three steps of subgraph dynamic expansion in proposed design flow: (a) Obtain the net bounding box before routing, (b) Estimate the initial coverage that provides most nets a sufficiently large subgraph to route, and (c) Perform boundary detection to trigger the (d) Dynamic expansion to guarantee that the routing subgraph is eventually large enough.

Note that the parallel routing can guarantee deterministic results, although it produces different results from original serial PathFinder routing algorithm. Assuming that there is a sequential version of our parallel approach that applies the same subgraph dynamic expansion strategy, it is obvious that single-net parallelization is sequential equivalent, and we will propose a multi-net parallelization that is also sequential equivalent.

3 Subgraph Dynamic Expansion

The computational kernel in FPGA routing is a solver for the single-source shortest path (SSSP) problem, usually using Dijkstra's algorithm or A* search.[1] Note that the fundamental data structure in both algorithms is a priority queue, which causes contentions and bottlenecks in a GPU implementation. Therefore, most existing literature and publicly-available solvers for the GPU-accelerated SSSP algorithm are based on the Bellman-Ford algorithm for a greater parallelism and speedup, although the worst-case sequential time complexity of the Bellman-Ford algorithm is higher than the Dijkstra's algorithm.

To take full advantage of the existing GPU-based SSSP solvers for FPGA routing, our basic idea is to alleviate its time complexity by reducing the problem size. In general, there are at least two approaches to doing so:

1) One is to perform global routing in a coarsened routing graph.
2) The other is to restrict the search scope for the SSSP algorithm.

The global routing approach is useful for ASIC routing, but it is less effective for FPGA routing [30]. It may be possible to obtain a pseudo-rectilinear structure from the FPGA routing graph by clustering the routing segment nodes inside the same channel. Taking Fig. 2 as an example, one may cluster the nodes a and b, c and d, e and f, g and h, respectively, so that the clustered nodes form a rectilinear structure for global routing. However, this conversion is not accurate to model the congestion, because it cannot distinguish the congestion cost of the segment nodes in the same channel. According to our analysis of the final routing results of the PathFinder algorithm across several benchmarks, we observe that the routing cost of some segment nodes in the same channel differ significantly. The maximum difference (e.g., 24) of the routing cost for the nodes in the same channel usually greater than the mean (e.g., 9) plus one standard deviation (e.g., 10) among the nodes with non-zero costs. Therefore, the global routing approach is not the best choice to reduce the problem size, and we resort to the other approach by restricting the routing scope.

To ensure the correctness and convergence of FPGA routing algorithm, we propose the method of *subgraph dynamic expansion* to limit the search space. It contains three essential steps to mitigate the disadvantages of the GPU-friendly Bellman-Ford algorithm.

- *Step 1:* subgraph extraction, which is implemented efficiently based on a labeling system that relates the coordinates to the routing resource nodes.
- *Step 2:* initial coverage, which is preprocessed by analyzing the routing results of existing circuits. Our estimation of the initial routing subgraphs provides sufficient routing nodes (e.g., for 98.5 percent nets in existing circuits), so that each net only needs to explore its routing tree in a small subgraph instead of the overall routing graph.
- *Step 3:* dynamic expansion, which is complementary to initial coverage using a detection strategy to adaptively expand the routing subgraph in a next routing iteration until a feasible solution is found.

Notice that the key idea of subgraph dynamic expansion is to estimate and find a large-enough routing subgraph for every net to obtain its own feasible route. In the initial coverage, we first determine the initial routing subgraphs to cover a significant portion of nets, and then we perform dynamic expansion in case that a few nets need a larger subgraph to find their legal routing trees. This method effectively bounds the number of nodes during the routing exploration, and thus alleviates the complexity overhead of the Bellman-Ford algorithm compared to the Dijkstra's algorithm.

The usage of subgraph dynamic expansion in proposed design flow is illustrated in Fig. 4. We first begin by a straightforward estimation of the routing subgraph is the one within the bounding box of *source*, *sink1* and *sink2* in Fig. 4a. Using the bounding box to estimate the initial subgraph does not provide enough routing resources in many cases, and thus, We then determine a good-enough subgraph at the initial coverage stage, as shown in Fig. 4b. The static estimation is unlikely 100 percent accurate, and a detour path outside the initial coverage may be necessary for a legal routing solution. So we apply a dynamic strategy to expand the subgraph, whenever a detour path touching the boundary of the current routing subgraph is detected,

1. Dijkstra's algorithm and A* search have similar data structures and algorithmic flow. Thus, we only mention Dijkstra's algorithm to compare with the GPU-friendly Bellman-Ford algorithm in the rest of this paper for conciseness.

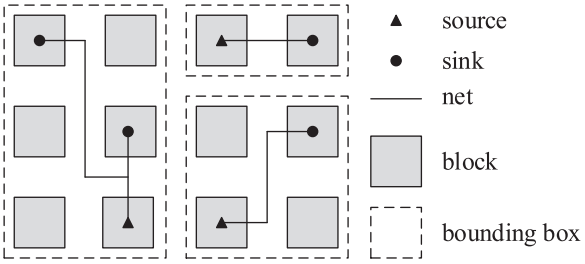

Fig. 5. Routing a single net in bounding box of the subgraph which can be extracted by the terminal nodes (source and sinks).

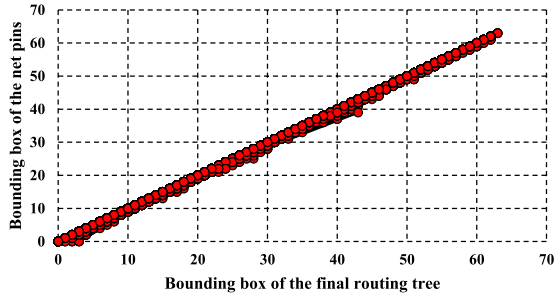

Fig. 6. The bounding box of the final routing tree is only slightly larger than the bounding box of the net pins for Δ_C.

as illustrated in Fig. 4c. Finally, we can find a routing solution inside the estimated and expanded subgraph as shown in Fig. 4d. Evaluations show that this approach effectively reduces the problem size without affecting the quality of the routing solution.

3.1 Subgraph Extraction

As discussed earlier, due to the limitations of the global routing approach for FPGAs, we resort to the method of restricting the search space (i.e., the routing subgraph) for the GPU-based SSSP solvers. The ideal restricted search space with the minimum number of routing resource nodes should let an SSSP algorithm generate the same or similar result as in the original search space. However, this very ideal case is non-trivial to obtain. In addition, extracting an irregular subgraph requires some timing-consuming graph traversals that affect the efficiency. Thus, we relax the ideal routing resource subgraph to be a NET-SPECIFIC BOUNDING BOX, which contains sufficient routing resource nodes for the given-net routing. In the following, we present the details how to extract a routing resource subgraph inside a bounding box.

We make the following assumptions for the proposed subgraph extraction. The entire routing graph consists of pin nodes and segment nodes. A pin node corresponding to a pin of a logic block is assigned with the placement coordinates of this logic block. For every segment node, there exists an edge connecting to a pin node, and this segment node shares the same coordinates with its neighboring pin node. Since some segment nodes are neighbors of two or more pin nodes with different coordinates, these segment nodes have multiple coordinates. Note that these coordinates of routing resources including nodes and edges are given after placement. Further, with the coordinates of source and sink nodes, we can determine the bounding box of subgraph and enable the single-net routing in the small box rather than entire graph.

Fig. 5 shows the subgraph corresponding to a bounding box in the FPGA routing region includes all the pin nodes and segment nodes with coordinates inside this box, as well as the routing edges between these nodes. Given any box in the FPGA routing graph, we can efficiently determine the subgraph in the box defined above. In the next two sections, we will discuss how to determine the size of such box for any given net and guarantee the convergence of the routing algorithm.

3.2 Initial Coverage

The routing subgraph defined in the previous section reduces the problem size. The next question is how to find the dimension of the box for the subgraph extraction, given any net to be routed.

As discussed earlier, to ensure the correctness and convergence of the routing algorithm, the box should contain sufficient routing resources for a given net, and its size is expected to be as small as possible. A simple choice is the minimum bounding box of the pins in a given net, which is available before routing. But this simple choice rarely provides enough routing resources. Another choice is the minimum bounding box of the final routing tree, which is of course only available after the routing finishes. This choice provides sufficient routing resources but is impractical and not implementable. To get a good-enough initial subgraph before routing, we propose the *initial coverage* in proposed router to estimate the initial boxes to cover a significant portion of nets with enough routing resources. The idea of this estimation is based on the statistical data from existing routed circuits.

An important observation is that for most nets, the size of the bounding box of the final routing tree is only slightly greater than the bounding box of the net pins, as illustrated in Fig. 6. Based on this empirical relation based on existing routed circuits, we can statistically estimate the size of the boxes for the routing subgraphs during the initial coverage. The box of a given net in the initial coverage is expanded from the four sides of the bounding boxes of net pins by a distance of Δ_C, as illustrated in Fig. 4b. The estimation of Δ_C relates to the FPGA size, as well as a user-defined percentage of coverage.

Here we describe an example flow to estimate Δ_C given a few circuits with known routing solutions. The distance Δ_C is the difference in the left, right, top and bottom boundary coordinates between the bounding box of the net pins and the bounding box of its final routing tree, which can be collected from these existing circuits. If the user-defined percentage of coverage is 98.5 percent, we find the smallest coverage that is not less than 98.5 percent of the nets in every given circuit. We observe that by dividing this value of Δ_C by the FPGA array size, we obtain a similar ratio, 0.021 on average, among many circuits. We call this ratio the *initial expansion factor*. Examples of this initial expansion factor are shown in Table 1.

Therefore, given a new circuit, we can multiply the FPGA array size by the initial expansion factor and round it up to the next integer to obtain Δ_C for the initial coverage. By applying this rule, we can estimate how much we should expand the bounding box to be the initial coverage when constructing of the initial routing resource subgraphs.

TABLE 1
An Example of the Initial Expansion Factor of 0.021 for a 98.5 Percent Coverage

Bench.	Δ_C	array	$\Delta_C\sqrt{\text{array}}$	coverage
diffeq2	1	34×34	0.029	100%
mkDelayW.	1	48×48	0.021	99.8%
blob_mer.	1	51×51	0.020	100%
mkPKtMer.	1	58×58	0.017	100%
or1200	1	65×65	0.015	100%
LU8PEEng	1	53×53	0.019	99.3%
bgm	2	73×73	0.027	99.5%
mcml	2	101×101	0.020	98.7%
average	—	—	0.021	—

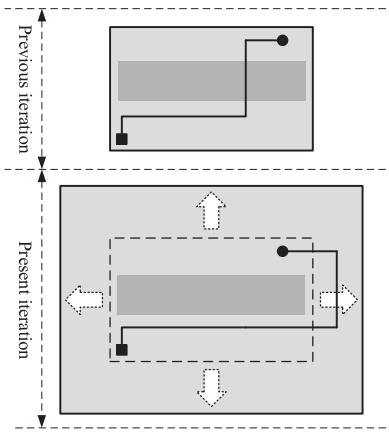

Fig. 7. The details of dynamic expansion of a single net box at each iteration.

3.3 Dynamic Expansion

Though the initial coverage can provide enough routing resources for most nets, there are still some outliers. A simple fix is to expand the subgraphs continuously to ensure sufficient routing resources eventually. However, such strategy will increase the routing time due to some unnecessarily large subgraphs. In GPU-friendly router, we use the *dynamic expansion* strategy, which contains a detection method to expand a subgraph only when necessary.

Dynamic expansion is based on a boundary detection strategy, as shown in Fig. 4c, which is used to decide whether we will continue to expand the box size for the routing subgraph of a net. A net is likely to use more routing resources when its routing tree occupies a node located on the boundary of the current routing region of subgraph. Once detected, we expand the bounding box of its routing subgraph on the four sides by a distance of Δ_D[2] in the next routing iteration. With this boundary detection strategy in dynamic expansion, this GPU-friendly router can converge using a similar number of iterations as the original PathFinder algorithm. Fig. 7 shows the dynamic expansion process of routing resource box of a single net during routing iteration. In previous iteration, a single net can not find a feasible path due to the congestion happens. With the expansion in the following iteration, the single net can detour to find a legal path in larger subgraph box.

2. This parameter is empirically set to one, which is sufficient to find a legal routing solution according to our experiments.

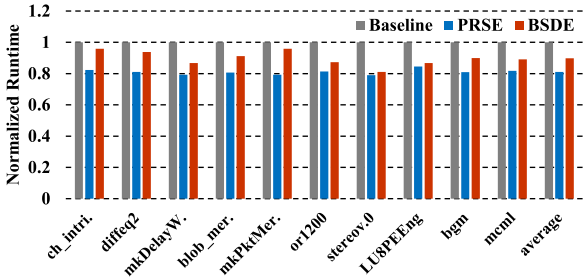

Fig. 8. Comparisons of the sequential routing time between the baseline and PRSE/BSDE.

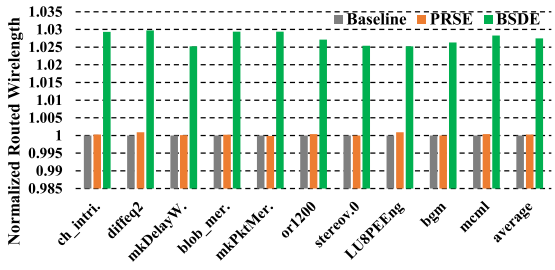

Fig. 9. Comparisons of the wirelength degradation between the baseline and PRSE/BSDE.

The subgraph dynamic expansion consists of subgraph extraction, initial coverage, and dynamic expansion. Its main purpose is to reduce the problem size to mitigate the worst-case time complexity of the Bellman-Ford algorithm, which is GPU-friendly and has efficient GPU-based implementations. Before discussing the GPU acceleration in the next section, here we evaluate the impact of subgraph dynamic expansion on the routing quality and the routing time, as well as the impact of replacing the Dijkstra's algorithm by the Bellman-Ford algorithm. Three approaches are evaluated and compared, including: the original PathFinder router (Baseline), the original PathFinder router optimized with subgraph dynamic expansion (PRSE), and the original PathFinder router optimized using the Bellman-Ford algorithm and subgraph dynamic expansion (BSDE). Note that the kernel of original PathFinder router is based on Dijkstra's algorithm.

Fig. 8 shows the normalized routing time of these two serial routing approaches. Benefiting from the subgraph dynamic expansion, all of them can reduce the routing time and the PRSE is faster than the BSDE approach. Though the worst-case time complexity of the Bellman-Ford algorithm is greater than the Dijkstra's algorithm, the runtime of BSDE approach is obviously better than the baseline with a negligible impact on the routed wirelength. Fig. 9 reports the normalized routed wirelength of the two approaches, where the routed wirelength is increased by about 2.7 percent on average using BSDE approach, and detailed explanation will be presented in Section 5.2.

Figs. 8 and 9 illustrate the effectiveness of the Bellman-Ford algorithm combined with subgraph dynamic expansion for FPGA routing. This computational kernel is GPU-friendly and is, therefore, a good candidate for the GPU-accelerated FPGA routing.

4 GPU-Accelerated Routing Exploration

In this section, we first explore the GPU parallelization of Bellman-Ford algorithm for single-net routing. We focus on

```
def static_node_parallelization:
    idx = get_tid ( )
    my_node = nodes[idx]
    if check ( my_node ) :
        compute ( my_node )
        *done_ptr = false

def main:
    done = false
    while not done:
        done = true
        static_node_parallelization<<<N>>> ( nodes )
```

```
def dynamic_node_parallelization:
    while idx = wl.pull ( ):
        my_node = nodes[idx]
        compute ( my_node )
    for all neighbors of my_node:
        if check ( neighbor ):
            wl.push (idx )

def main:
    init_wl<<<N>>> ( nodes, wl )
    dynamic_node_parallelization<<<M>>> ( nodes, wl )
```

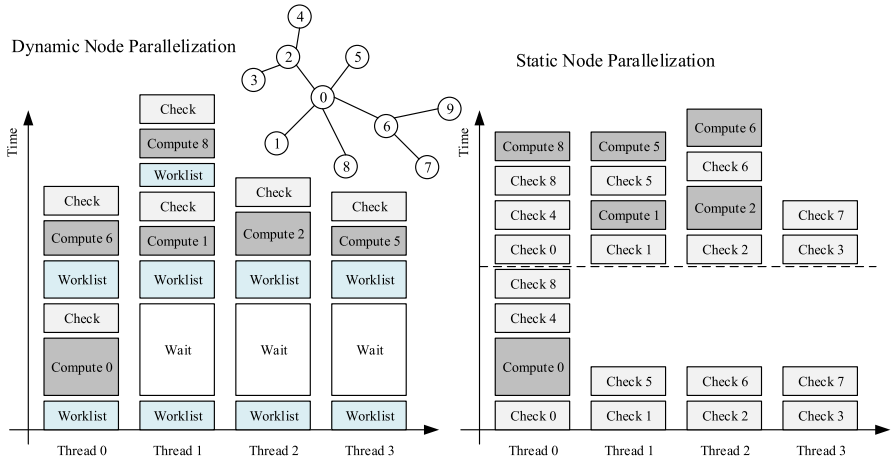

Fig. 10. The Implementations of example kernel on static and dynamic node parallelization, where N is the number of nodes, M is the maximum number of hardware threads, and wl is a worklist class. In static node parallelization, we are based on thread index to determine the work and all the nodes are visited whether they are active or not. Note that the number of threads spawned is equal to the number of nodes. In dynamic node parallelization, we access the shared worklist to determine the work and only active nodes are visited. Note that the number of threads spawned is equal to the number of hardware threads.

three strategies: static node parallelization, dynamic node parallelization, and dynamic edge parallelization. According to the net features, we then design a hybrid approach enabling a better parallelism in routing a single net. We finally explore the multi-net parallelization to obtain a further speedup for FPGA routing.

4.1 Fine-Grained Single-Net Parallelization

Considering that the specific structure of routing resource graphs, the previous experiences of parallelization strategies for general graphs cannot be directly applied to the routing resource graphs, especially for a small subgraph. Thus, it is very necessary for single-net routing to explore and examine the effectiveness of different kinds of parallelization in the GPU-based Bellman-Ford algorithm.

Typically, there are two practical approaches enabling the irregular Bellman-Ford algorithm to GPU, one is static node parallelization and the other is dynamic node parallelization. Fig. 10 gives the implementation details on static and dynamic node parallelization. Both of them iteratively apply a set of operators on a subset of elements in the data structure which are referred to as active nodes. 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 work required for the algorithm to progress and can generate more work by activating inactive nodes. Execution completes when all nodes are inactive. As shown in Fig. 10, there are two examples about the static and dynamic node parallelization.

In addition, we also focus on dynamic edge parallelization and exploit Merrill's optimization method [33] for better parallelism. Thus, we have the following three approaches to accelerate the Bellman-Ford algorithm with GPU for single-net routing.

1. Static node parallelization (SNP), where every node, no matter active or not, is assigned to a thread to process in parallel.
2. Dynamic node parallelization (DNP), where only the active nodes, i.e., the nodes whose known shortest distances to the source node have recently been changed, are assigned to threads to process in parallel.
3. Dynamic edge parallelization (DEP), which is similar to DNP but considers assigning active edges to threads instead of active nodes. This approach is optimized by Merrill's method [33].

In SNP, every thread first checks whether its responsible node is active. If active, it then applies the compute operator to update the known shortest path of the active node in each superstep. Every kernel execution on the GPU forms a superstep, and the kernel is invoked again in the next superstep when active nodes exist. All nodes, including active and inactive, are statically assigned to the threads through a block decomposition during the parallelization in every superstep.

In DNP, a centralized worklist[3] with atomic memory operation (AMO) is used to manage the dynamic parallelism [32]. First, an initialization step pre-checks all 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 an active node from the worklist using AMO and then applies the compute operator to the corresponding active node. The newly activated nodes are pushed onto the worklist with AMOs such that only active nodes will be visited in the next iteration. This process is repeated until the worklist becomes empty. Compared with SNP, DNP exposes more parallelism and improves the efficiency by mapping threads to useful computation work. However, the DNP also present its weakness with high memory contention when accessing a shared worklist.

A better implementation can be obtained by exploring the DEP using Merrill's method [33]. Besides focusing on the edges instead of nodes, DEP use a prefix scan to allocate a chunk of memory for each thread to maintain the active nodes so that it relieves the contention of atomic accesses by avoiding AMO. These chunks of memory are assigned to the CUDA blocks, which work in parallel to check the edges in their assigned chunks, using various heuristics to trade-off time and space for a high throughput.

3. The worklist is a variant of priority queue.

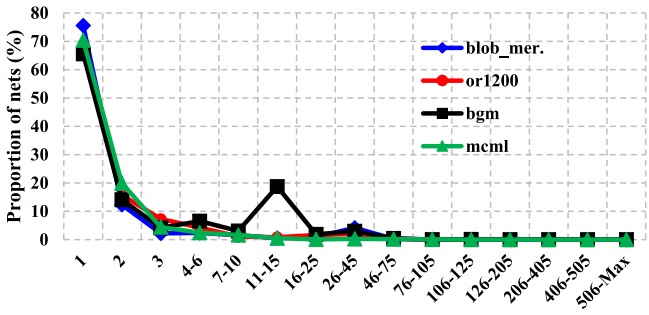

Fig. 11. Percentage of nets with different number of sinks.

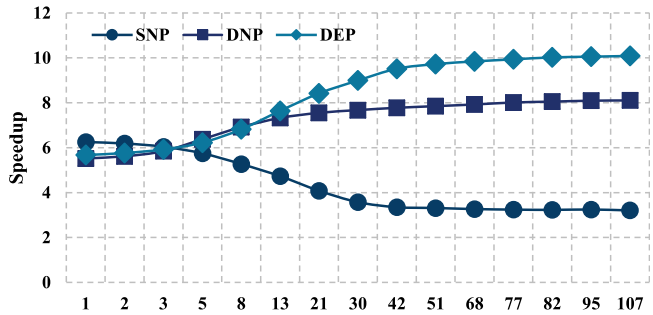

Fig. 13. Speedup with respect to the HPWL.

4.2 Fine-Grained Hybrid Approach

Either static or dynamic parallelism has its merits and demerits. Although slower than DNP and DEP for large graphs, the static SNP method achieves greater speedup for the low-fanout nets than the dynamic DNP and DEP methods. Our explanation is that the routing subgraph of a low-fanout net only has a small number of routing nodes and thus, the static assignment of GPU threads to these nodes is efficiently executed on the hardware. In single-net parallel routing, we present a hybrid approach to exploit the merits of both the static and dynamic parallelisms.

To seek higher speedup for FPGA routing, we analyze how different attributes of a net affect the runtime of different methods. Fig. 11 shows the percentage of the nets with a different number of sinks on four representative circuits. The number of low-fanout nets is significantly higher than the high-fanout nets. Thus, there is an opportunity to improve the speedup using a net-specific parallelization strategy. Fig. 12 shows the speedup using the SNP, DNP, and DEP methods of the nets with a different number of sinks in the *or 1200* circuit. For this specific circuit, we observe that the static SNP method is better than the dynamic methods for the nets with fewer than three sinks, and the dynamic DEP method is superior to the others for the nets with more than thirteen sinks. Moreover, we explore the speedups from the SNP, DNP and DEP methods with respect to the half perimeter wirelength (HPWL) of the nets and observe similar results, as shown in Fig. 13. We also observe similar patterns for other circuits by conducting the same set of experiments.

These results reveal the opportunity to combine different methods to improve the speedup. We propose an efficient hybrid approach that uses SNP for the nets with less than or equal to three sinks and uses DEP for the remaining nets. Though there is a possible gain using the DNP method for the nets with a moderate amount of sinks, these nets only contribute to a small percentage of runtime, and we simply apply DEP instead. We will present experimental evaluations in Section 5.2, which will show that our hybrid approach is as efficient as an "optimal" combination of SNP, DNP, and DEP.

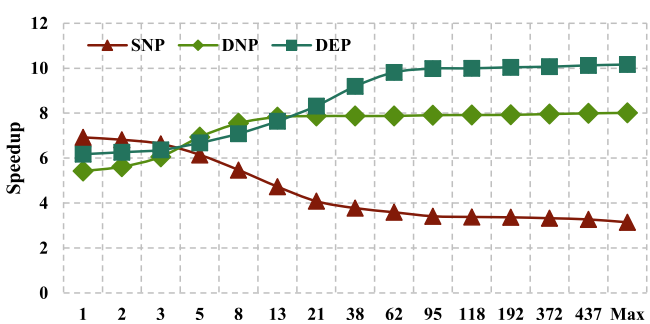

Fig. 12. Speedup with respect to the number of sinks.

4.3 Coarse-Grained Multi-Net Parallelization

In the above sections, we exploit SNP, DNP, DEP, and their hybrid approach to explore the fine-grained node- and edge-level parallelization for a single-net routing on a GPU platform. In this section, we attempt to explore the coarse-grained net-level parallelization for multi-net routing to obtain a further speedup on the GPU. Specifically, during the multi-net parallelization, we can maintain the equivalent routing results as the single-net routing parallelization.

According to previous works [35], the routing order of the nets will affect the final routing quality. To exploit net-level parallelism while maintaining the deterministic results, we need to satisfy a requirement that the routing results are equivalent to the single-net routing according to the original net order. Maintaining the original net order is also to ensure the convergence of the iterative parallel routing and avoid the degradation in the routing quality. At the meanwhile, only the independent nets can be partitioned for multi-net parallelization. Thus we give the definitions as follows:

Definition 1 (Independent Net). *There is a independent net if the bounding box of its subgraph does not overlap with the bounding box of other net, i.e.,*

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

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

In terms of each net k_i, we have an unique bounding box b_i, which can be determined according to the coordinates of the terminal nodes of net routing subgraph mentioned in Section 3.1. This subgraph box is used to limit the routing scope of single net and for each subgraph box 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 2 (Net Order). *A net k_i is the basic processing element in coarse-grained multi-net parallel routing. All of the nets can be labeled to form a set $N = \{k_1, k_2, \ldots, k_n\}$, and we partition these nets in an increasing net order as $k_1, k_2, \ldots, k_n$.*

Notably, the increasing net order is the same to the original net order of serial VPR router. With the requirements of

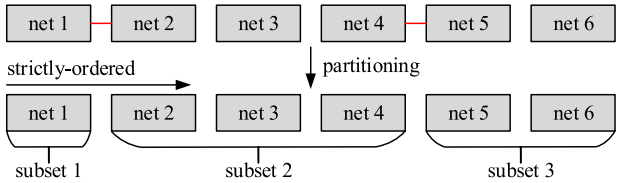

Fig. 14. The dynamic programming-based partitioning forms a series of subsets and each subset has multiple independent nets for parallel routing. The red connection denotes that there is a overlap between two adjacent boxes.

net order, we start to explore the partitioning of multiple nets to enable that our coarse-grained parallel router does not have an impact on the final routing results.

In addition, to make our parallel router has the equivalent results as the corresponding serial router, we introduce a strictly-ordered feature to the partitioning exploration. It is meaning that when finishing the partitioning of multiple nets, we have the strictly-ordered subsets and in each subset we have the independent nets which can be routed in parallel.

Definition 3 (Strictly-Ordered). *For a set of nets* $N = \{k_1, k_2, \ldots, k_n\}$ *through partitioning, we have a series of strictly-ordered subsets* $M = \{s_1, s_2, \ldots, s_m\}$ *if* $s_p = \{k_{i_p}, k_{i_p+1}, \ldots, k_{i_{p+1}-1}\}$, *where* $1 = i_1 < i_2 < \cdots < i_{m+1} = n + 1$.

We have these strictly-ordered subsets, because for $i < j$, every net in subset s_i is routed before a net in subset s_j according to the original net order. In terms of the generated subsets, we perform the parallel routing of the nets in the first subset s_1, and then after synchronization, we perform the parallel routing of the nets in the second subset s_2, and until to finish the parallel routing.

Note that the strictly-ordered feature is a necessary condition in partitioning and the independent nets is a sufficient condition in parallelization, both of which enabling the parallel router to generate the equivalent routing results as the serial router. With the above definitions, we formally formulate this partitioning problem to implement this parallel router.

Problem Formulation. Given a set of nets $N = \{k_1, k_2, \ldots, k_n\}$, our goal is to find a partition to generate a series of strictly-ordered subsets $M = \{s_1, s_2, \ldots, s_m\}$ and each subset consists of independent nets so as to minimize the total parallel routing time.

Consider that the total routing time depends heavily on the number of subsets, this motivates us to minimize the number of subsets in partitioning to obtain the minimum parallel routing time. Further, we can solve this partitioning problem in a dynamic programming algorithm. All of the nets $k_1, k_2, \ldots, k_n$ are partitioned into a series of subsets $s_1, s_2, \ldots, s_m$, respectively. Note that $k_1 + k_2 + \cdots + k_n = n$ and the original net order are maintained to these partitioned subsets.

Fig. 14 demonstrates an example about the dynamic programming-based partitioning approach. According to the original net order, all of the nets are partitioned into several strictly-ordered subsets and in each subset there are independent nets. We perform the parallel routing of multiple independent nets in same subset, thereby having the equivalent results as the above single-net parallel routing. Based on the bounding box of subgraph of each net, we can judge whether two adjacent boxes are overlaps or not, further determining the dependencies between two adjacent nets to implement the partitioning of multiple nets. Thus, this partitioning approach is effective and precise in parallel routing.

TABLE 2
Notations for the Partitioning Problem

Notation	Description
$F[j][i]$	The feasible indicator whether these elements $k_j, k_{j+1}, \ldots, k_i$ are independent nets or not.
$D[i]$	The minimum number of subsets for the nets from k_1 to k_i under the requirements of strictly-ordered and independent properties.

The bounding box of net subgraph and its expansion approach mentioned in Section 3 provide an effective detection to determine whether two adjacent nets are dependent or not. The partitioning approach starts to route the first net in current subset, and gradually adds the next independent nets to the subset, until there is a overlap with the next adjacent nets in this subset. It is obvious that the nets in same subset can be routed in parallel while obtaining the same results as the single-net parallel routing. Therefore, the multi-net parallelization can further provide a significant speedup for FPGA routing.

The dynamic programming-based partitioning algorithm is performed in a quadratic time. We analyze the details about the time complexity of the partitioning algorithm as follows. Table 2 shows the critical notations involved to the partitioning algorithm.

Specifically

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

This algorithm consists of two steps: precomputation and dynamic programming. In precomputation, we adopt the simple pair-wise testing algorithm to calculate the value of $F[j][i]$ and the worst-case complexity is a quadratic time. But in practice, it is very fast to perform this simple algorithm due to that the number of strictly-ordered subsets is very small.

According to the value of $F[j][i]$, we start to perform the dynamic programming algorithm. The minimal number of subsets of the first i net satisfies

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

The solution to the problem is $D[N]$. It is obvious that for given $F[i][j]$ value, its time complexity of calculating the value of $D[N]$ is $O(N^2)$. Thus, the time complexity of the overall partitioning 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 algorithm is to perform the partitioning of nets and the total number of nets is relative small. Further, the j value that needs to be enumerated is significantly smaller than the total number of nets. Thus, it

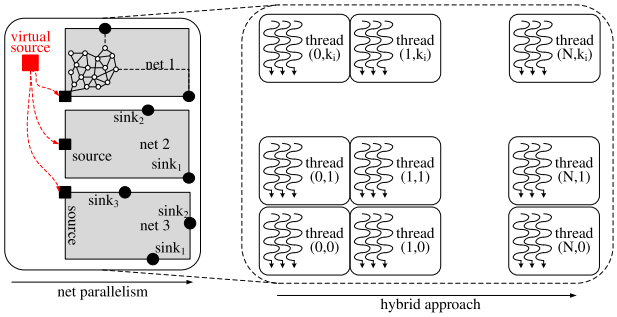

Fig. 15. Concurrent routing on GPU.

is very fast to perform the dynamic programming-based partitioning algorithm. In addition, due to the sparseness of routing resource graph, there will be much fewer strictly-ordered subsets than the independent nets used to be routed in parallel. Thus, it is effective to employ the partitioning algorithm to our multi-net parallel router.

The above partitioning algorithm generates a series of *strictly-ordered* subsets to maintain the final quality, combined with the *independent* nets in each subset, to obtain the same results as the single-net parallel routing. Moreover, our parallel router is clearly scalability based on strictly-ordered partitioning.

Fig. 15 shows that the multiple independent nets are routed concurrently on GPU. To unify the single-net and multi-net routing, a virtual source node is imposed to directly connect to the actual source nodes of the independent nets. Thus, the independent nets can be combined into a single pseudo net, which can be routed on GPU leveraging the single-net acceleration techniques as discussed previously. Fig. 16 explains the sources of speedup when routing multiple nets in parallel. The vertical axis reveals how the size of the worklist, which is the number of edges processed concurrently in the GPU-based SSSP algorithm, varies with respect to the execution time. The speedup comes from the reduction of the filling time of the worklist at the beginning and the evicting time near the end. So that when we route multiple nets as a single pseudo net, some overheads in these two phrases are eliminated to further improve the available speedup.

5 Experimental Study

In this section, we evaluate all the proposed parallel approaches introduced in the previous sections in terms of

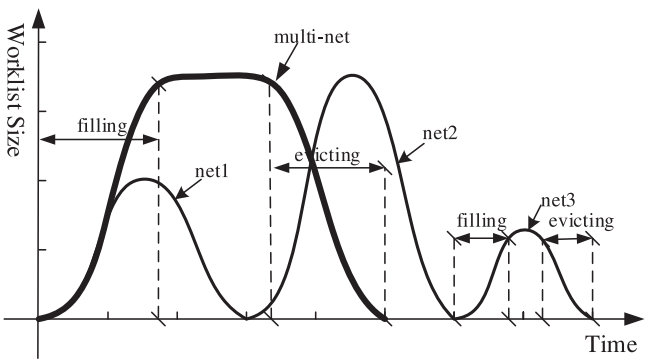

Fig. 16. Sources of speedup.

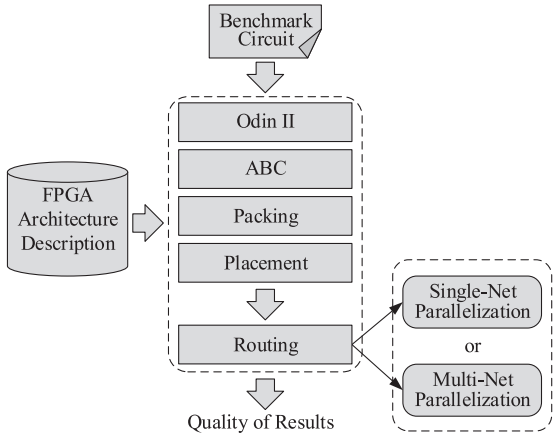

Fig. 17. The experimental CAD flow.

quality of results and runtime of the routing. We also demonstrate the effectiveness of the proposed parallel router when processing on a routing resource graph with increasing scales.

5.1 Experimental Setup

We adopt the state-of-the-art VTR 7.0 CAD compilation tool flow [43] in these experiments as shown in Fig. 17. This flow takes a benchmark Verilog circuit and an FPGA architecture description file as input. The flow maps the circuit to the architecture described in that file, then outputs statistics about that final mapping. We use Odin II for elaboration, ABC for logic synthesis, AAPack for packing, and VPR for placement and routing. VPR is left at default values [43] and the original router will be accelerated in proposed parallel approaches.

We use the commonly used VTR 7.0 benchmarks for our experiments. The VTR 7.0 benchmarks are a standard set of Verilog circuits that come from a variety of different applications including computer vision, medical, math, soft processors, etc. These circuits contain heterogeneous elements, such as memories and multipliers, which differ from older small-scale MCNC benchmarks. Table 3 summarizes the characteristics of these circuits, including the application

TABLE 3
Benchmark Summary

Circuit	Domain	Architecture	Size	Nets	CLBs
ch_int.	Memory Init	k4_N4_90nm	20x20	788	497
sha	Cryptog	k4_N4_90nm	29x29	1946	866
boundt.	Ray Trace	k4_N4_90nm	19x19	2380	724
diffe.2	Math	k4_N4_90nm	34x34	3710	1296
diffe.1	Math	k4_N4_90nm	35x35	3953	1450
mkDela.	Packet Proc	k4_N4_90nm	48x48	5224	1554
blob_m.	Image Proc	k4_N4_90nm	51x51	6606	2702
mkSMAd.	Packet Proc	k4_N4_90nm	53x53	7154	3126
mkPKtM.	Packet Proc	k4_N4_90nm	58x58	7474	3767
or1200	Soft Proc	k4_N4_90nm	65x65	8078	3648
stereo.0	Comp Visi	k6_N10_40nm	39x39	9312	1492
stereo.1	Comp Visi	k6_N10_40nm	39x39	13523	1401
LU8PEE.	Math	k6_N10_40nm	53x53	16278	2373
bgm	Finance	k6_N10_40nm	73x73	27853	4225
stere.2	Comp Visi	k6_N10_40nm	86x86	36479	2802
mcml	Med Phys	k6_N10_40nm	101x101	81282	7934

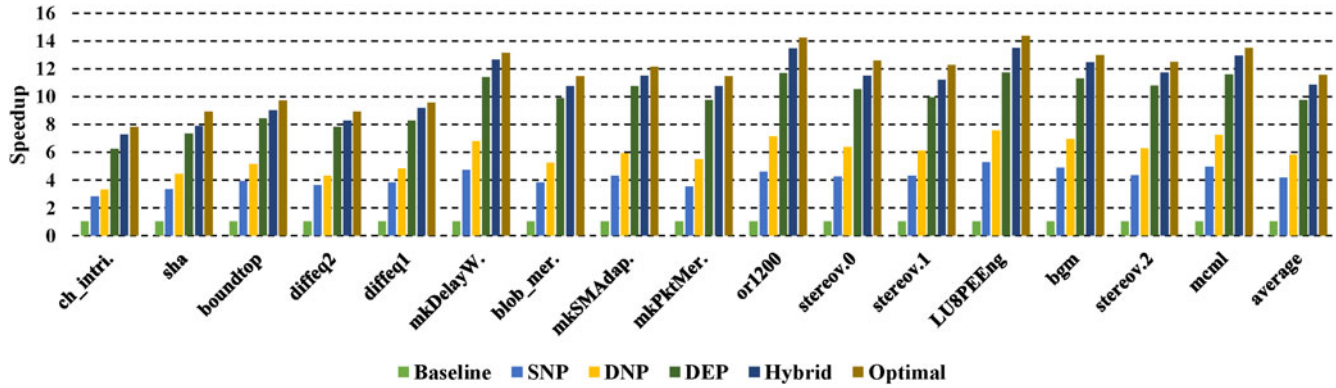

Fig. 18. The single-net parallelization on GPU using SNP, DNP, DEP, and Hybrid approach.

domain, the array size of the routing region generated, the number of nets, and the number of configuration logic blocks (CLBs) used. Across all runs, each circuit is routed using a channel width of $1.3\times$ the minimum channel width needed by VPR router, following the same configurations as in the previous works [13], [14], [15].

We conduct all experiments on a Linux server with a 8-core Intel Xeon CPU at 2.2 GHz and 32 GB shared memory, equipped with a Tesla K40c GPU having 2,880 cores in 15 streaming multiprocessors and 12 GB video memory. The baseline for comparisons is the original VPR 7.0 router [43], which is a sequential program implemented in C. The academic VPR 7.0 router is faster than commercial router [7] and it is always used for comparisons in parallel routing research. Some of the GPU implementations of the SSSP algorithm are adapted from the source code in the LonestarGPU collection [31].

The state-of-the-art serial VPR 7.0 router has two different optimization goals, one is routability-driven router and the other is timing-driven router. The former is used to optimize the total routed wirelength and the latter is used to optimize the critical path delay. In this paper we leverage the proposed parallel approaches to accelerate the routability-driven router and evaluate the available speedup and total routed wirelength. Note that these two kinds of serial routers have the same data structure and algorithmic flow, thus our proposed parallel approaches are suitable for the timing-driven router as well.

5.2 Runtime and Available Speedups

Fig. 18 shows the runtime and achieved speedup using four different parallelization techniques to accelerate single-net routing in one by one. The runtime and speedup of each benchmark are shown in each column. The leftmost column is the baseline, and the next four columns show the runtime and available speedup of SNP, DNP, DEP, and the hybrid approach. To illustrate the effectiveness of our hybrid approach, we include the available speedups of an optimal hybrid approach in the last column. The average speedups over all benchmarks are shown in the last row. On average we achieve a speedup of about $4.15\times$, $5.82\times$, $9.75\times$, and $10.86\times$ with the SNP, DNP, DEP, and Hybrid, respectively. The runtime of the optimal hybrid approach is estimated by summing up the fastest possible runtime of each net using either SNP, DNP, or DEP, assuming there is an oracle to predict the optimal selection. The $10.86\times$ speedup of our hybrid approach is only slightly less than the speedup of the optimal hybrid approach, $11.57\times$ on average.

We can observe in Fig. 18 that the hybrid approach in our router achieves more speedup than other parallel methods of SNP, DNP, and DEP. The reason is that our hybrid approach invokes SNP to route a significant number of low-fanout nets, and routes the timing-consuming multi-sink nets using DEP. Moreover, the hybrid approach is compatible with the previous coarse-grained parallel methods [13], [15] to achieve a further speedup.

Fig. 19 presents the available speedups of parallelization of multi-net routing on GPU using the hybrid approach. It can be seen that this approach produces an average speedup of about $21.53\times$ using a single GPU. This is a $3.94\times$ improvement over the recent fine-grained parallel router [14], and a $3.05\times$ enhancement over the recent coarse-grained parallel router [15]. We do obtain notable speedups for the four largest benchmarks on the right in the figure, owing to that more independent nets can be combined into a single pseudo net, further resulting in more acceleration on GPU. Moreover, this approach is promising to be extended with the multi-GPU parallelization to improve the speedup greatly [33].

Finally, we list the speedups of previous coarse-grained and fine-grained parallel FPGA routers, compared to the sequential VPR router in Fig. 20. By taking advantage of GPU acceleration, we proposed parallel router can provide significant speedups. It is the first work to accelerate FPGA routing using GPU. It is also the first work to achieve near $20\times$ speedup for the FPGA routing problem.

According to the above experimental results, we can conclude that the subgraph dynamic expansion enables the

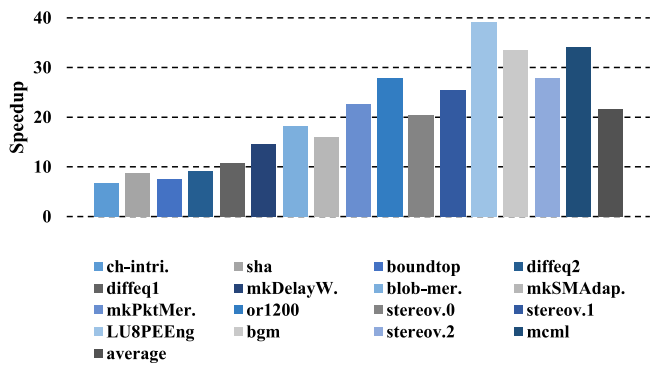

Fig. 19. The multi-net parallelization on GPU using the hybrid approach.

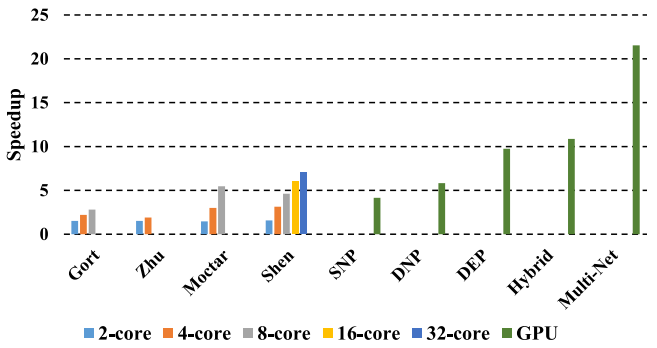

Fig. 20. Speedups of SNP, DNP, DEP, Hybrid, and multi-net parallel routing approach compared to existing works.

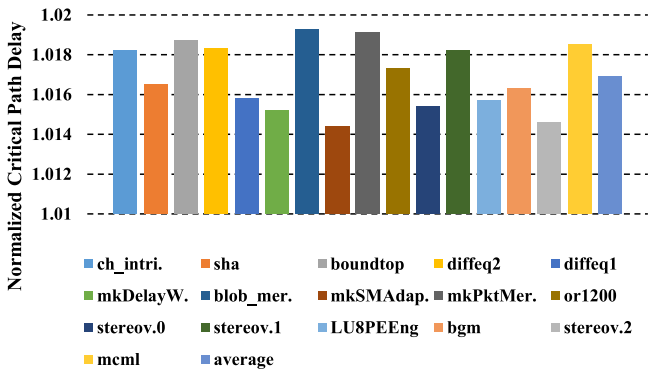

Fig. 22. Impacts on the critical path delay using the multi-net parallelization.

single-net parallel routing approaches on GPU to provide a good speedup, especially for the proposed hybrid approach. With the dynamic programming-based partitioning, we leverage the hybrid approach to perform the parallel routing of multiple nets on GPU to obtain a further improvement to the available speedup. In addition, our final parallel router is very faster than the previous parallel routers in terms of total speedup.

5.3 Quality of Results

In this experiment, we evaluate the final routing quality of the multi-net parallel routing approach due to that its speedup is faster than other proposed parallel routing approaches. Our approach is used for routability-driven router to evaluate the total routed wirelength. Moreover, our approach is also integrated into the timing-driven router to evaluate the critical path delay. Figs. 21 and 22 give the quality of results about the total routed wirelength and critical path delay, respectively.

In Fig. 21, we compare the routing quality of multi-net parallelization with the routability-driven serial VPR 7.0 router regarding the total routed wirelength. This router only introduces about 2.73 percent degradation in wirelength on average, compared to the original VPR router. In Fig. 22, we present the results of critical path delay when comparing the multi-net parallel router with the timing-driven serial VPR 7.0 router. On average, there is about 1.68 percent degradation in critical path delay. Their degradations mainly come from the multi-sink nets. The original VPR router performs the Dijkstra's algorithm many times to obtain a final routing tree. Our router directly combines the shortest paths from the single source to the multiple sinks in a single round of the Bellman-Ford algorithm into a full routing tree.

The multi-net parallel router introduces about 2.73 and 1.68 percent degradations in wirelength and critical path delay respectively when comparing to the original VPR 7.0 router. This impact is negligible for the scenarios such as the synthesis of reconfigurable FPGA accelerators in datacenters and fast design iterations in early design stages. In the former case, the delay degradation is insignificant compared to the orders of magnitude speedups introduced by FPGA acceleration. And in the latter case, the design quality is not as important as the design productivity.

5.4 Parallel Scalability

With FPGA integration density scales, routing resource graph will continue to grow every generation. Thus, a scalable routing algorithm becomes essential to provide similar speedup when the size of routing graph grows. Here, we evaluate the multi-net parallelization on large-scale routing graphs. We construct synthetic designs on large-scale routing graphs based on given benchmarks. The locations of the sources and sinks of the nets in a benchmark are linearly stretched when we extend the FPGA array size from 100×100 to 1000×1000. The routing resource graph for the 1000×1000 array size is at the same scale as the largest benchmark in Titan [7].

Fig. 23 gives the speedups obtained on three representative benchmarks over the sequential VPR router. We achieve the best speedup for the FPGA array size around 500×500. While the speedup decreases slowly when the FPGA array size grows, the multi-net parallel router still

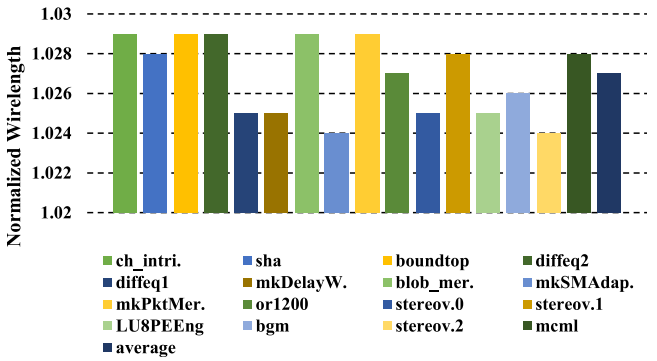

Fig. 21. Impacts on the total routed wirelength using the multi-net parallelization.

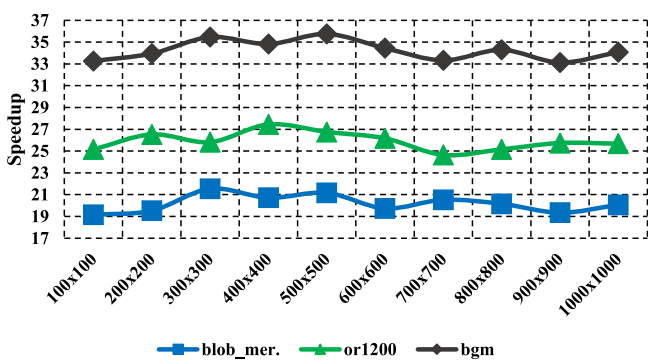

Fig. 23. Scalability analysis with different FPGA array sizes.

scales well. The scalability of parallel router is attributed to two reasons: 1) The subgraph dynamic expansion strategy effectively reduces the problem size. 2) The GPU-based SSSP is a variant of the Bellman-Ford algorithm, which makes use of the worklist that behaves similarly to a queue. Actually, in many practical cases, the Bellman-Ford algorithm converges faster than the worst-case analysis [37], [38], and there exist examples [39] that the queue-based Bellman-Ford algorithm spends less computation than the Dijkstra's algorithm. Here we provide another example, the variant of queue-based Bellman-Ford algorithm, is practical for FPGA routing. We observe the same trends for other benchmarks and thus, these results can indicate that the proposed parallel router is promising to maintain a similar speedup for large-scale routing graphs.

6 Related Work

Most of the previous works on parallelizing FPGA routing are motivated by the acceleration of the overall synthesis time. The first parallel PathFinder algorithm is proposed by Chan and Schlag [11]. By modeling the routing problem into a graph or hypergraph matching problem, they analyze how and when the history and present congestion cost of the PathFinder routing algorithm should be synchronized across the processors to ensure convergence while improving parallelism [34]. Although their method is highly sensitive to the order of the nets to be routed, it is still an open problem to determine the best net ordering [35]. Quite noticeable is that a speedup of 2.5× is attainable using three processors on a distributed cluster.

The fine-grained parallelization avoids the influences of net ordering in the PathFinder routing algorithm. Dehon et al. [36] propose the design of a hardware accelerator for FPGA routing. Their simulations predict a speedup of up to three orders of magnitude over PathFinder with 5-25 percent loss in solution quality. Zhu et al. [12] attempt to partition 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 percent loss in solution quality. And then Moctar and Brisk [14] explore the dynamic parallelism using the Galois API. They achieve a good speedup of 5.4× using eight threads. Recently, Hoo et al. [16] propose a fully parallel router based on Lagrangian relaxation to decompose the original routing problem into independent subproblems. This approach can produce an average speedup of 7× using eight threads, compared to its sequential version.

While the coarse-grained parallelism is sensitive to the net order, Gort and Anderson [13] propose a deterministic parallel PathFinder routing algorithm. They partition the nets into subsets, and these subsets are routed in parallel with an efficient synchronization scheme to guarantee the deterministic results. Although they did not emphasize the scalability, it is the first deterministic parallel routing algorithm and achieves a 2.8× speedup using eight cores. Another deterministic parallel router is proposed by Shen and Luo [15], using a partitioning-based parallel routing method. They leverage a dynamic programming algorithm to determine the optimal recursive partitioning strategy. Although it degrades the quality of routed wirelength, the parallel router exploits more parallelism and can scale to a 32-core cluster with an average speedup of 7×.

Design reuse is also an attractive strategy to reduce the FPGA synthesis time. HMFlow [40] attempts to reuse precompiled logic, called hard macros, to reduce the routing time. BPR [41] then focuses on larger macros to permit greater speedups. Intermediate fabrics [42] exploits the reuse of precompiled virtual FPGA architecture to reduce the synthesis time. The integration of parallelization and design reuse is a promising direction for an ultra-fast FPGA router.

In this paper, we explore subgraph dynamic expansion combined with GPU-accelerated SSSP algorithm for parallel FPGA routing. Notice that some of the previous approaches can be adopted in proposed approaches to achieve a greater speedup. Moreover, This is the first work to leverage GPU to accelerate FPGA routing efficiently.

7 Conclusion and Future Work

In this paper, we explore GPU-accelerated routing for FPGAs. We first use the approach of subgraph dynamic expansion to obtain convergent routing results with a reduced problem size. This approach enables the efficient application of the GPU-friendly Bellman-Ford algorithm to replace the Dijkstra's algorithm in PathFinder. We empirically observe that for most nets, the bounding box of the final routing tree is only slightly larger than the bounding box of the net pins so that we can estimate the routing subgraphs for most nets effectively. The process of dynamic expansion is also helpful to obtain a routing subgraph with sufficient routing resources for the exceptional cases. We then perform systematic experiments and comparisons among different GPU accelerations of the Bellman-Ford algorithm, including the SNP, the DNP, and the DEP approaches. We point out that we can combine the static SNP and dynamic DEP methods for a greater speedup and exploit the multi-net parallelism, where we achieve an average of 21.53× speedup on GPU. Although our approach increases the wirelength and critical path delay by about 2.73 and 1.68 percent respectively, it is still meaningful for fast design iterations and the design of FPGA-based accelerators.

In this paper, We demonstrate the effectiveness of GPU acceleration for FPGA routing. Specifically, the subgraph dynamic expansion is also applicable to the problem size reduction for other shortest path algorithms. In the future, our work can be enhanced in multiple ways, including 1) using multiple GPUs for further acceleration, 2) improving the accuracy of the subgraph estimation (e.g., a congestion-aware estimation) in the initial coverage, 3) exploring the techniques to extract independent nets for the acceleration of multi-net routing on GPU, and 4) leveraging similar ideas to explore the parallelization techniques for an FPGA-accelerated FPGA router.

Acknowledgments

We appreciate the insightful comments and feedbacks from anonymous reviewers. This work is partly supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61433019 and No. 61802446. This work is also partly supported by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant No. 2016ZT06D211.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. Int. Symp. Comput. Archit.*, 2011, pp. 365–376.

[2] K. Atasu, et al., "Accelerating text analytics queries on reconfigurable platforms," in *Proc. 4th Workshop Intersections Comput. Archit. Reconfigurable Logic (CARL)*, 2015.

[3] P. K. Gupta, "Xeon+FPGA platform for the data center," in *Proc. 4th Workshop Intersections Comput. Archit. Reconfigurable Logic (CARL)*, 2015.

[4] T. Brewer, "Convey's acceleration of the Memcached and Imagemagick applications," in *Proc. 4th Workshop Intersections Comput. Archit. Reconfigurable Logic (CARL)*, 2015.

[5] A. Putnam, et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. Int. Symp. Comput. Archit.*, 2014, pp. 13–24.

[6] K. Ovtcharov, et al., "Accelerating deep convolutional neural networks using specialized hardware," in *White Paper*, 2015.

[7] 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," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, 2015, Art. no. 10.

[8] C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, vol. EC-10, no. 3, pp. 346–365, Sep. 1961.

[9] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 1995, pp. 111–117.

[10] B. Catanzaro, K. Keutzer, and B. Su, "Parallelizaing CAD: A timely research agenda for EDA," in *Proc. ACM Annu. Des. Autom. Conf.*, 2008, pp. 12–17.

[11] P. Chan and M. Schlag, "Acceleration of an FPGA router," in *Proc. IEEE Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 1997, pp. 175–181.

[12] C. Zhu, J. Wang, and J. Lai, "A novel net-partition-based multi-threaded FPGA routing method," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl.*, 2013, pp. 1–4.

[13] M. Gort and J. Anderson, "Deterministic multi-core parallel routing for FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl.*, 2010, pp. 78–86.

[14] Y. Moctar and P. Brisk, "Parallel FPGA routing based on the operator formulation," in *Proc. ACM Annu. Des. Autom. Conf.*, 2014, pp. 1–6.

[15] M. Shen and G. Luo, "Accelerate FPGA routing with parallel recursive partitioning," in *Proc. Int. Conf. Comput. Aided Des.*, 2015, pp. 118–125.

[16] C. Hoo, A. Kumar, and Y. Ha, "ParaLaR: A parallel FPGA router based on lagrangian relaxation," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl.*, 2015, pp. 1–6.

[17] J. Croix and S. Khatri, "Introduction to GPU programming for EDA," in *Proc. Int. Conf. Comput. Aided Des.*, 2009, pp. 276–280.

[18] N. Kapre and D. Ye, "GPU-Accelerated high-level synthesis for Bitwidth optimization of FPGA datapaths," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2016, pp. 185–194.

[19] D. Chen and D. Singh, "Parallelizing FPGA technology mapping using graphics processing units," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl.*, 2010, pp. 125–132.

[20] C. Fobel and D. Stacey, "GPU-Accelerated wire-length estimation for FPGA placement," in *Proc. Int. Conf. Appl. Accelerators High-Perform. Comput.*, 2011, pp. 14–23.

[21] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. Int. Conf. High Perform. Comput.*, 2007, pp. 197–208.

[22] U. Meyer and P. Sanders, "Delta-stepping: A parallel single source shortest path algorithm," in *Proc. Annu. Eur. Symp. Algorithms*, 1998, pp. 393–404.

[23] Y. Deng and S. Mu, "Electronic design automation with graphic processors: A survey," in *Proc. Int. Conf. Found. Trends Electron. Des. Autom.*, Jan. 2013, vol. 7, pp. 1–176.

[24] A. Bleiweiss, "GPU accelerated pathfinding," in *Proc. Int. Symp. Graph. Hardware*, 2008, pp. 65–74.

[25] Y. Han, K. Chakraborty, and S. Roy, "A global router on GPU architecture," in *Proc. Int. Conf. Comput. Des.*, 2013, pp. 78–84.

[26] F. Busato and N. Bombieri, "An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2222–2233, Aug. 2016.

[27] A. Davidson, S. Baxter, M. Garland, and J. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2014, pp. 349–359.

[28] A. DeHon, et al., "GraphStep: A system architecture for sparse-graph algorithm," in *Proc. IEEE Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2006, pp. 143–151.

[29] M. Shen and G. Luo, "Corolla: GPU-accelerated FPGA routing based on subgraph dynamic expansion," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 105–114.

[30] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Berlin, Germany: Springer, Feb. 1999, ISBN 0-7923-8460-1.

[31] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. Annu. Int. Symp. Workload Characterization*, 2012, pp. 141–151.

[32] K. Pingali, et al., "The tao of parallelism in algorithms," in *Proc. Int. Symp. Program. Language Des. Implementation*, 2011, pp. 12–25.

[33] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. Int. Symp. Principles Practice Parallel Program.*, 2012, pp. 117–128.

[34] P. Chan, M. Schlag, C. Ebeling, and L. McMurchie , "Distributed-memory parallel routing for field-programmable gate arrays," in *Proc. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 8, pp. 850–862, Aug. 2000.

[35] R. Rubin and A. Dehon, "Timing-driven pathfinder pathology and remediation: Quantifying and reducing delays noise in VPR-pathfinder," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2011, pp. 173–176.

[36] A. Dehon, R. Huang, and J. Wawrzynek, "Hardware-assisted fast routing," in *Proc. IEEE Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2002, pp. 205–215.

[37] S. Zhou, C. Chelmis, and V. Prasanna, "Accelerating large-scale single-source shortest path on FPGA," in *Proc. Int. Symp. Parallel Distrib. Process. Symp. Workshop*, 2015, pp. 129–136.

[38] A. Dandalis, A. Mei, and V. Prasanna, "Domain specific mapping for solving graph problems on reconfigurable devices," in *Proc. Int. Symp. Parallel Distrib. Process.*, 1999, pp. 652–660.

[39] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Reading, MA, USA: Addison-Wesley, 2011, ISBN: 032157351X.

[40] C. Lavin, et al., "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping," in *Proc. IEEE Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2011, pp. 117–124.

[41] J. Coole and G. Stitt, "BPR: Fast FPGA placement and routing using macroblocks," in *Proc. Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2012, pp. 275–284.

[42] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Proc. Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2010, pp. 13–22.

[43] J. Rose, et al., "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, 2014, Art. no. 6.

Minghua Shen received the PhD degree in computer science from the Peking University, in 2017. He is currently an associate researcher with the School of Data and Computer Science, Sun Yat-sen University, China. His research interests include FPGA synthesis, heterogeneous and parallel computing, cyber-physical systems. He is a member of the IEEE and ACM.

Guojie Luo received the BS degree in computer science from Peking University, Beijing, China, in 2005, and the MS and PhD degrees in computer science from UCLA, in 2008 and 2011, respectively. He received the 2013 ACM SIGDA Outstanding PhD Dissertation Award in electronic design automation and the 10-year Retrospective Most Influential Paper Award at ASPDAC 2017. He is currently an associate professor with the School of EECS, Peking University. His research interests include electronic design automation, heterogeneous computing with FPGAs and emerging devices, and medical imaging analytics. He is a member of the IEEE and ACM.

Nong Xiao received the BS and PhD degrees in computer science from the College of Computer, National University of Defense Technology (NUDT), China, in 1990 and 1996, respectively. He is currently a professor with the School of Data and Computer Science, Sun Yat-sen University. His current research interests include network parallel computing, large-scale storage system, and computer architecture. He has more than 160 publications to his credit in journals and international conferences, including the *IEEE Transactions on Services Computing*, the *IEEE Transactions on Multimedia*, the *Journal of Parallel and Distributed Computing*, the *Journal of Computer Science and Technology*, the *International Journal of High Performance Computing Applications*, ICCAD, MIDDLEWARE, MSST, IPDPS, CLUSTER, SYSTOR, and MASCOTS. He is a senior member of the IEEE and a member of the ACM.