2018 IEEE 36th International Conference on Computer Design

Fine-Grained Parallel Routing for FPGAs with Selective Expansion

Minghua Shen and Nong Xiao
School of Data and Computer Science, Sun Yat-Sen University, China
Email:{shenmh6,xiaon6}@mail.sysu.edu.cn

***Abstract*—FPGAs are reconfigurable architectures that can offer large performance and energy improvements over general purpose processors. However, compiling an application design onto the underlying FPGA device takes commonly too much time to allow efficient design turnaround times, significantly hindering designer productivity. Routing is always a very timing-consuming and critical process in FPGA compilation flow. To reduce FPGA routing time, parallel techniques have become more popular in recent years.**

In this paper, we propose a fine-grained GPU-based parallel routing approach that enables high concurrent single-net parallel routing for large-scale FPGAs. The proposed approach is novel in three ways. The first is Scheme-1. We route a single net only on its own bounding box rather than entire routing resource graph and then selectively expand its bounding box to make sure that single-net routing has a feasible solution. The second is Scheme-2. We impose a GPU thread to each node and perform path searching in dynamic programming algorithm on all the nodes simultaneously for single-net parallel routing on a single GPU. Note that these nodes are distributed in the bounding box of single net. The third is optimized Scheme-2. We attempt to partition single-net routing box into several sub-boxes, each of which is processed in a shared memory of GPU to enable high scalability and parallelism.

Our evaluation with ten large designs from the academic VTR benchmark suite shows that our approach of combining the approximation of VPR 7.0 router (Scheme-1) and its parallelization (optimized Scheme-2) provides a 1.57x speedup improvement compared to the best parallelization-only approach of the original VPR 7.0 router, though at a small quality deterioration.

***Index Terms*—FPGA, FPGA CAD, Routing, Parallel Routing, Selective Expansion.**

I. Introduction

With the slowing down of Dennard scaling, computing systems are steadily moving towards the use of application or domain specific accelerators for higher performance and energy efficiency [3]. Accelerators customize data and control flows to meet a domain of applications, thereby reducing some flexible overheads of general purpose processors. However, specializing the accelerators in the form of dedicated ASICs is very expensive due to the high non-recurring engineering (NRE) costs for design and fabrication, as well as the high deployment and turnaround times [4]. Thus, it is impractical to employ the ASIC-based accelerators for all, except for the most ubiquitous applications.

FPGAs are flexible architectures able to compensate the high NRE costs by equipping configurable logic blocks with a programmable interconnect switch to implement these customized data and control paths. In FPGAs, these customizable paths can be configured at the bit level, thereby allowing designers to prototype an arbitrary digital logic system and take fully advantage of architectural feature to satisfy the arbitrary precision computation. This flexibility has not only resulted in a number of successful commercial FPGA-based accelerators deployed in data centers [5], [6], [7], but also leaded to a substantial reduction of the economic risk of developing hardware accelerators. It also explains the rise in the popularity of the FPGAs.

Unfortunately, the flexibility of the FPGA comes at the cost of compilation inefficiencies. Each time the application design must be compiled to an FPGA configuration. Compilation has several stages, each of which involves solving NP-complete problems. While existing FPGA manufacturers and academics have developed heuristics to approximate an optimal solution, they still need to take hours, days or even weeks to complete the compilation, depending on the sizes of the implemented designs and the target circuits. Companies such as Altera [1] and Xilinx [2] have developed new FPGA products containing millions of logic cells or logic elements such that large-scale designs can be efficient mapped to the underlying FPGA device, further increasing the compilation time. Long compilation time hinders designer productivity since compilation will be performed several times to fix bugs and test performance. Thus it is essential to reduce compilation time while preserving its quality of results.

Routing is always a very time-consuming and critical process in FPGA compilation flow [20]. The problem of routing FPGA can be stated simply as that of assigning nets of given circuits to routing resources of the target architecture in order to successfully route all nets while achieving a given overall performance. This is a challenging problem to route a design on an FPGA because of the relative scarcity of routing resources, both wires and connection points. This can lead either to slow implementations caused by long wiring paths that avoid congestion or a failure to route all nets. It is difficult to produce the best quality within an acceptable runtime. At present, most of FPGA routers use iterative algorithms that try to converge progressively a feasible solution. During each iteration, all the nets are typically ripped-up and re-routed one by one.

One of the most popular of such routers is PathFinder [8], which attempts to balance the competing goals of eliminating congestion and minimizing delay of critical paths. PathFinder enables all the nets to be routed to achieve close to the optimal

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00093

performance, thereby becoming a very attractive algorithm in academic and commercial communities. For example, the two largest FPGA companies, Altera and Xilinx, employ a variant of the PathFinder algorithm in their commercial routers [9], [10]. Moreover, PathFinder is also used in the publicly-available VPR FPGA placement and routing framework [21], which we parallelize on GPU platform. However, PathFinder algorithm is naturally sequential and parallelization is non-trivial. This is because PathFinder imposes congestion costs to routing resources and the costs are incremented after routing each net, the costs seen by a net depend on the resources used by previous nets. As a result, the costs introduce dependencies among nets and pose challenges to effective parallelization of the PathFinder algorithm.

There have several recent efforts [17], [14], [16], [18] in parallelizing PathFinder routing for FPGAs. We present a novel parallel routing approach that can exploit selective expansion for single-net routing, and also explores fine-grained GPU-based single-net parallel routing techniques that aim to make parallel router higher concurrency. Our work mainly makes the following contributions:

- We propose Scheme-1, a selective expansion approach that exploits the dependencies to expand the net bounding box in a single direction. It not only reduces the runtime of serial routing algorithm but also provides the potential opportunity to parallelize the single-net routing.
- We propose Scheme-2, a fine-grained GPU-based parallel routing approach that regards each node as a GPU thread and performs path searching in dynamic programming fashion on all of the nodes simultaneously. Note that all of these nodes only originate from the net bounding box rather than entire resource graph.
- We demonstrate promising speedups in routing time for a set of large designs from the academic VTR benchmark suite. Notably, our combination of the approximated VPR 7.0 router (Scheme-1) and its parallelization (optimized Scheme-2) has a 1.57x speedup improvement compared to the best parallelization-only approach of the original VPR 7.0 router with a small quality deterioration.

The rest of this paper is organized as follows. Section II gives background and Section III analyzes motivation. Section IV details the proposed parallel routing approach. Section V reports the results. Section VI draws conclusions.

II. Background

A. FPGA Routing

The routing resources of an FPGA can be modeled as a directed graph, where node represents a wire or a pin and edge represents a programmable connection point between a pin and a wire segment, or a programmable routing switch between two wire segments. A single net has one source and one or more sinks, and routing a single net is to search a legal path to connect the source to all its sinks. The paths for different nets must be disjoint to prevent short circuits.

By imposing the costs to the nodes of a graph, we can optimize the routing problem. The cost on a node is related to the congestion on the corresponding resources, and we apply a graph-based search technique to find a feasible solution by updating the costs based on available resources. Typically, the routing process consists of path searching and congestion elimination. During path searching, all of the feasible paths will be explored to select the most suitable path for each net. After path searching, some routing resources exist congestions, enabling the nets to be ripped-up and re-routed for congestion elimination.

B. PathFinder Algorithm

PathFinder [8], a negotiation-based routing algorithm, is a well-known FPGA router. PathFinder consists of two major parts, global router and signal router as shown in Fig. 1. During PathFinder routing, global router invokes signal router to route a net and receives the path information of single-net routing from signal router. The signal router employs the breadth-first search to find the shortest path measured by cost. The global router iteratively reduce the congested resources until to find a feasible routing solution. In each iteration, the global router rips-up and re-routes all nets to liberate the resources that have been used by the previous routed nets.

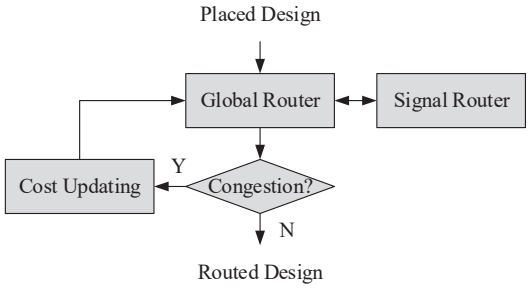

Fig. 1. PathFinder routing flow.

To determine the order of nets routing through the congested resources, PathFinder imposes a congestion cost function to release the congested resources iteratively. The cost function of using a given node n in the i-th iteration is given by

$$c_n = (d_n + h_n^{(i)}) * p_n$$

where d_n is the base cost for delay, $h_n^{(i)}$ is related to the history of congestion on resource node n during previous iterations of the global router, and p_n is the number of other nets presently using resource node n in the signal router. With $h_n^{(i)}$, non-congested nets can release a resource for congested nets, while with p_n, congested nets can negotiate with each other for using a resource.

In routing a single net our problem is formulated as follows.

FPGA single-net routing problem: Given a graph $G = (N, E)$ in which every node is associated with a positive routing cost, the objective is to find a path in graph G that connects source to sinks such that the total routing cost of the path is minimal.

In this paper we parallelize the signal router that accelerates single-net routing for FPGAs.

C. Related Work

In recent years, a large number of parallel routing works have been proposed. We mention here only those most closely related to our work and Table. I gives a classification of related works.

TABLE I

Name	CPU Platform	GPU Platform
Fine-Grained Fashion	[12], [18]	Our Work
Coarse-Grained Fashion	[11], [13], [15], [16]	[14], [17]

Fine-grained CPU-based parallel routers explore single-net parallel routing for FPGAs. In [12], the single-net is parallel routed by using lock based expansion operator and software transactional memory based priority queue. They can expose high parallelism but the overhead of lock acquisition and rollback reduces the speedup of parallel router. In [18], the multi-sink net is decomposed into single-sink nets, and their bounding boxes are shrunk to increase the number of nets that can be routed in parallel.

Coarse-grained CPU-based parallel routers explore parallel routing of multiple nets for FPGAs. In [11], [13], all of the nets are partitioned into several subsets, and these subsets are processing in parallel. In [15], all of the nets are parallelized in speculative parallelism and communication overhead is reduced by encoding the routing path in a space-efficient manner. In [16], the synchronous and asynchronous parallelism is coordinated to multi-net routing. To expose high parallelism, they dynamically partition the nets and optimize the rip-up and re-route stage for good speedup.

Coarse-grained GPU-based parallel routers exploit GPU parallelism to accelerate multi-net routing. In [14], they enable GPU-based speculative parallelism such that each resource can be mapped to a GPU thread and multiple nets are processed in parallel. Another work [17] leverages dependency-aware partitioning to assign the nets into dependent subsets. The nets in same subset can be parallel routed using a GPU and the subsets are processed in serial. Moreover, this partitioning approach can be implemented by a dynamic programming technique for an optimal solution.

III. Motivations

Our approach consists of two single-net routing improvement schemes (Scheme-1 and Scheme-2). These schemes are proposed based on two main motivations presented in this section. Specifically, Motivation-1 and Motivation-2 below motivate for Scheme-1 and Scheme-2, respectively.

Motivation-1: The net bounding box can be expanded in a single direction.

In routing a single net, the bounding box can limit the net searching scope and makes the single-net routing only in its own small box rather than entire routing region. Each net is surrounded by its rectangular bounding box, containing the net terminals and the routing resources as shown in Fig. 2. During the routing, the bounding box is commonly used to determine the dependencies between nets and if the bounding boxes of two nets are overlap, the possibility of congestions between the nets is high and they need to rip-up and re-route in next iteration. Notably, each net has an unique bounding box in routing.

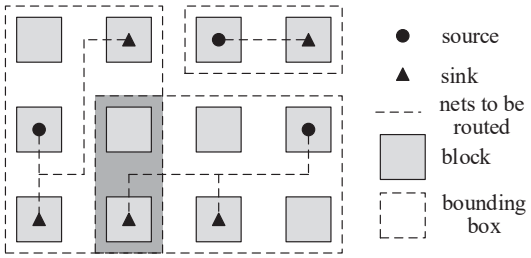

Fig. 2. Net bounding box. The overlap of bounding boxes of two nets is highlighted in gray.

Bounding box expansion is used to guarantee the routing resources within the bounding box are sufficient to implement a legal routing path. The failure to find a legal routing path within the bounding box will activate the bounding box to be expanded such that the net is re-routed again in expanded bounding box during the next iteration. Thus, based on the results of bounding box expansion, the dependency between nets can be determined before the next routing iteration.

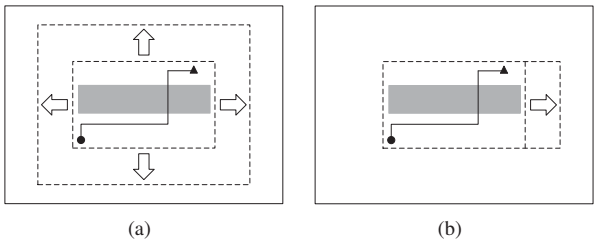

Fig. 3. Example of routing a net with a vertical congestion resource. (a) previous box expansion. (b) Congestion-based selective expansion.

The prior techniques [14], [17], [21] expand the net bounding box continuously in four directions with routing iteration proceeds as shown in Fig. 3(a). However, these schemes tend to over-expand, subsequently increasing the runtime. This also imposes the additional number of congested nets and that results in more congestions between nets due to some unnecessary boxes to be expanded.

The box expansion scheme based on current dependent state between nets has seldom been explored in previous literatures. For example, Fig. 3(a) shows a routing path has a vertical congestion resource. Existing expansion schemes expands the bounding box along both x and y coordinates to resolve the congested situations. Actually, the vertical expansion is unnecessary in Fig. 3(a) and the bounding box only needs to expand horizontally as shown in Fig. 3(b). Based on this motivation, a novel congestion-based selective expansion is presented to avoid the over expansion of net bounding box and reduce the runtime of single-net routing.

Motivation-2: A high degree of parallelism can be exposed in fine-grained single-net parallel routing.

Most of the work on parallel FPGA routing focuses on PathFinder routing algorithm and either global router or signal router is parallelized to accelerate the routing process. In global router the multiple nets are routed in parallel to expose the coarse-grained net-level parallelism while in signal router the single net is routed in parallel for fine-grained subnet-level parallelism. The coarse-grained net-level parallelism needs to analyze the dependencies between nets by examining the overlaps of bounding boxes of nets. Essentially nets without any overlaps are regarded as independent nets that can be extracted to be routed in parallel regardless of their original routing order.

However, the degree of parallelism is insufficient in the coarse-grained fashion. We analyze this effect through an experiment with a series of benchmarks from the state-of-the-art academic VTR benchmark suite [21]. We employ coarse-grained fashion into global router to calculate the exploitable net-level parallelism for each benchmark. In each iteration of global router, parallel nets will be extracted based on their dependencies. In parallel signal router, we employ fine-grained fashion to calculate the exploitable subnet-level parallelism. Conceptually this fashion can expose a higher degree of exploitable parallelism to the parallel router.

We assume that a single-sink subnet is defined as a routing task and this single-sink subnet is decomposed from the single net topology. Thus a single net has either one single-sink subnet or several single-sink subnets according to the number of sinks of the net. When exploring net-level parallelism, coarse-grained fashion follows the coherent principle, requiring that only one subnet belonging to a same net is allowed to be routed in parallel each time. Fine-grained fashion relaxes this principle and allows the parallel routing of multiple subnets from a single net, enabling the high subnet-level parallelism. Note that fine-grained fashion parallelizes a net routing one by one and it has the same number of iterations to the serial router.

To calculate the exploitable parallelism, we need to analyze the dependencies between nets to determine whether two nets can be routed in parallel or not. If there are no overlaps between the boxes of two nets in the current routing iteration, they are independnet nets and can be routed in parallel. Further, according to the box size of the current iteration and the default box expansion, we can determine the dependencies between nets in the following iteration, further determining the parallelly routable nets. Fig. 4 shows an example of how to determine the dependencies between two nets.

We follow the original net order of VPR 7.0 router, and based on the above described pairwise determination of net independence, we determine subsets of parallelly routable nets that respects that order to obtain serially equivalent results. To obtain the smallest possible parallel runtime, we need to partition the set of nets into a minimum number of such parallelly routable net subsets, where the subsets are routed sequentially per the original net order, while the nets in each subset are routed in parallel. We do so by using a dynamic-programming method to determine the minimum number of "stretches" of mutually independent nets (parallelly routable subsets) in the original order. This partitioning technique was proposed in previous work [17], where further details are available.

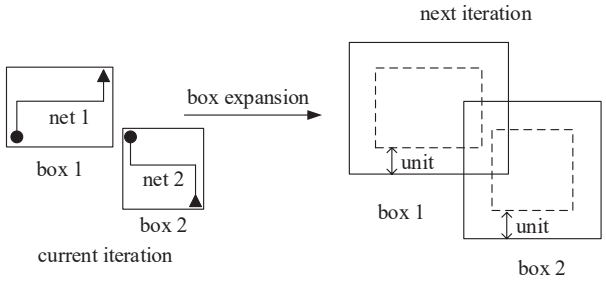

Fig. 4. An example to determine the dependencies between two nets. The net 1 and net 2 are independent and parallelly routable in the current iteration, but dependent and thus not parallelly routable in the next iteration.

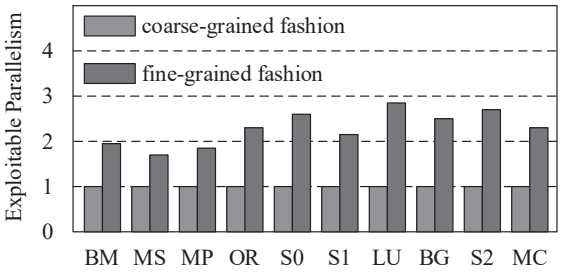

Fig. 5. Exploitable parallelism in state-of-the-art academic VTR benchmark suite. The y-axis is normalized to the coarse-grained fashion.

Assuming enough bandwidth to parallelize the routing of all the nets at each iteration, we quantify exploitable parallelism as N/M, where N is the total number of subnets[1] and M is the number of iterations to find a feasible solution. Fig. 5 shows the results of exploitable parallelism between coarse- and fine-grained fashions. The coarse-grained fashion shows the limited exploitable parallelism and such limitation further worsens in the large benchmarks. In contrast, the fine-grained fashion does not show a similar trend, demonstrating better concurrency in the capability of parallel nets extraction. Notably, we observe that up to 3× exploitable parallelism is exposed in fine-grained fashion.

IV. Our Approach

In this section we explain how our schemes work and discuss the implementation details of fine-grained GPU-based parallel routing approach we employ. Specifically, Scheme-1 exploit congestions between nets to expand the bounding box of each net and Scheme-2 leverage GPU to parallelize the single-net routing in its own box.

[1]Before parallel routing, the number of nets is known and the number of sinks of each net is also known, thus the total number of subnets is known.

A. Algorithmic Flow

We still use negotiation-based rip-up and re-route framework [8] such that our parallel router can converge a feasible solution with the limited number of iterations. Our proposed parallel routing flow is shown in Fig. 6.

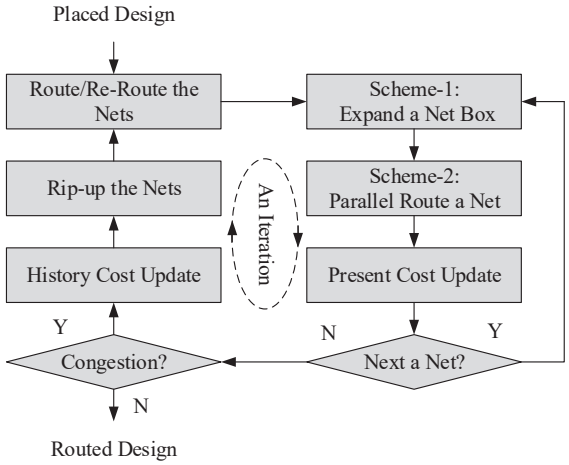

Fig. 6. Our parallel routing flow.

Our approach has two important schemes: Scheme-1 and Scheme-2. In Scheme-1, we enable each net to be routed in its bounding box rather than entire routing region in order to diminish the congestions between nets and reduce the path searching space. We select a single direction to expand the bounding box of a net in each iteration and with iteration proceeds, our approach is still able to converge a feasible solution. Notably, since the congestions between nets reduce, the number of iterations provided by our Scheme-1 is smaller than the number of iterations used by the original serial router. In Scheme-2, we parallelize the single-net routing on a GPU. We assign a GPU thread to each node and parallelly perform path searching on all of the nodes. Note that these nodes only locate in the bounding box of a net to be routed in parallel. Moreover, during parallel routing, we consider on-chip shared memory allocation combined with spatial partitioning for highly scalable parallel routing.

In iterative process, we have a correct judgment about the dependencies between nets before each iteration. We can also determine which nets will result in congestions with others in next iteration. This is because we can obtain such dependencies between nets by detecting which net boxes exist overlaps based on previous iteration. Combining with Scheme-1, we are able to know how size will be expanded for the bounding box of each net in the subsequent iteration. Moreover, we only expand the searching space of congested nets rather than all of the nets in each iteration.

B. Scheme-1: Congestion-Based Selective Expansion

Routing is an iterative algorithm that motivates us to exploit iterative property to analyze and speculate the congestions between nets and further improve the routing algorithm. When re-routing a single net in next iteration, we firstly trace the routing path of the net based on the previous iteration and analyze the number of congested resource nodes, locating in horizontal and vertical routing channels as shown in Fig. 2.

We assume that if the number of horizontal congested nodes is larger than the number of vertical congested nodes, the bounding box of the net will be expanded in vertical direction by σ units. On the contrary, the bounding box will be expanded in horizontal. If this results in a tie, the bounding box will be expanded randomly in vertical or horizontal. Expanding the bounding box of a net in a single direction can efficiently restrict the size of net box and that can reduce the number of overlaps among bounding boxes of nets. Thus, there are more independent nets to be produced in each iteration, thereby reducing the number of iterations.

Implementation Details. The initial bounding box is the minimum bounding box of the net to be routed in Fig. 7(a), and we expand the initial bounding box of each net in a σ unit in iterative process. Note that the expansion unit σ is set to 1 by default, which is the same as the previous works [14], [17], [21]. Since two opposite sides of bounding box may have different congestion states, we only expand a single side close to the non-congestion area and the other side is unnecessary to be expanded. Thus we need to analyze the expansion to each boundary of bounding box, respectively.

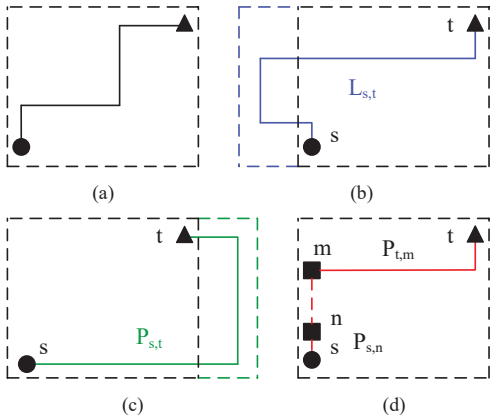

Fig. 7. Selective expansion. (a) Initial bounding box. (b) Left expansion and $L_{s,t}$ is expected to be across the left boundary of bounding box. (c) Right expansion and $P_{s,t}$ is identified. (d) The estimation cost on $L_{s,t}$ is the sum of the costs of $P_{s,n}$ and $P_{t,m}$ and manhattan distance $md(n,m)$.

We analyze each boundary to determine which boundary needs be expanded in the next iteration. Here we select the left boundary of the bounding box as an example to demonstrate the implementation details. Left boundary expansion implies that we have the potential to search a legal routing path $L_{s,t}$ on the left boundary of bounding box as shown in Fig. 7(b). Assuming that there also exist another routing path $P_{s,t}$ and its path cost is smaller than the cost of the current path $L_{s,t}$ as shown in Fig. 7(c). In other words, the routing cost of path $P_{s,t}$ is lower than the least routing cost of path $L_{s,t}$, thereby implying the left boundary expansion is unnecessary. However,

the least cost routing path $L_{s,t}$ is unknown due to that the area on the left boundary has not been explored yet.

We are therefore inspired to estimate the minimal cost $mcost$ for a single-net routing. The estimation cost $mcost$ is used to speculate whether the boundary of bounding box is necessary to be expanded. *Relative to the left routing path $L_{s,t}$, there exist another routing path $P_{s,t}$ and if its cost is smaller than $mcost_L$, the estimation cost of the left routing path $L_{s,t}$, the left boundary is unnecessary to be expanded at the following iteration.* The estimation cost for left routing path $L_{s,t}$ has the following function:

$$mcost_L = MIN_{m \in V_L, n \in V_L}\{d(s,n) + d(t,m) + md(n,m)\}$$

where V_L represents the set of nodes located on the left boundary of bounding box. $d(s,n)$ and $d(t,m)$ denote the least cost to the routing paths from s to n and from t to m, respectively. $md(n,m)$ is the manhattan distance between n and m. Note that both $d(s,n)$ and $d(t,m)$ are known values that have been computed by the previous routing in Fig. 7(d).

Implementation Flow. With congestion-based selective expansion techniques, when we start to re-route a net in the next iteration, the routing path of previous iteration is firstly traced to obtain the number of vertical and horizontal congestion nodes. If the number of vertical congestion nodes is larger, we will expand the left and right boundaries of bounding box in horizontal direction. If the net is not be routed in the previous iterations, the left and right boundaries are expanded in immediate. Otherwise, we are based on the above implementation detail to determine which boundary will be expanded.

Based on the deterministic expansion in bounding box of each net, we can know the bounding box size of each net and also determine which boxes exist overlaps before re-routing the net in parallel. Most importantly, every net with selective expansion has the deterministic searching space, enabling us to explore single-net parallel routing in a small searching space rather than entire routing space. In the following section, we introduce the single-net parallel routing on GPU platform.

C. Scheme-2: Fine-Grained Parallelization with GPU

The previous works on coarse-grained parallel routing [14], [17] have dealt with PathFinder routing algorithm on GPU architecture. We improve this idea by attempting to exploit the extreme parallelism of the GPU to perform fine-grained parallel routing. However, it is non-trivial due to GPU excels at the algorithms with regular memory access patterns. Since single-net routing is a graph-based searching algorithm, access patterns can be arbitrary depending on topology of the graph.

GPU Architecture. The fundamental building block of an NVIDIA GPU is the Streaming Multiprocessor (SM) and is shown in Fig. 8. Each SM contains 8 stream processors (SP), 1 double precision floating point unit (DP), and 2 special function units (SFU). Each SM executes groups of 32 threads referred to as a warp.

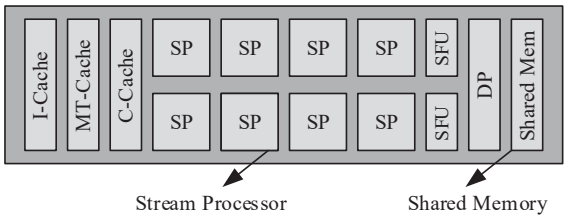

Fig. 8. Streaming Multiprocessor.

The SM contains a single program counter shared among all threads in the warp so these threads must execute the same instruction in lock-step. If there are control flow instructions that cause divergence between threads in the warp, then these threads must be serialized and cannot be executed completely in parallel. An important feature of the SM is a small amount of shared memory (16K bytes) that can be shared among the 8 SPs. The access time for this memory is typically just one clock cycle whereas the time to access off-chip DRAM can incur a penalty of hundreds of clock cycles. A modern GPU architecture consists of an array of SMs whose only method of communication is through the off-chip DRAM.

```
Input: 1) Graph topology of bounding box, G(V, E).
       2) Source and sink nodes of a single net, N(s, t_i).
       3) A routing cost from node m to node n, c(m, n).
Output: Minimum routing cost from source node s to node n
at iteration k, A[n, k].

Begin
  A[n, 0] = ∞, A[s, 0] = 0.
  Copy A from main memory into the off-chip DRAM of GPU.
  While (An unstable node exists) {
      ParaBegin
          A[n, k] = min{A[m, k-1] + c(m, n), A[n, k-1]}.
      ParaEnd
  }
  Copy A from the off-chip DRAM of GPU into main memory.
End
```

Fig. 9. Fine-grained parallel routing of a single net on GPU.

Fine-Grained Parallel Routing. We parallelize single-net routing in its own bounding box using a GPU. Each resource node of bounding box is regarded as a GPU thread and performs the path searching on all nodes of bounding box simultaneously. Note that path searching is implemented by dynamic programming algorithm. Fig. 9 lists the fine-grained parallel routing of a single net on GPU. The array A is used to store the minimum routing cost from source node s to each node n. Algorithm starts by copying the array A from main memory of the CPU to the off-chip DRAM of the GPU. Then each GPU thread employs dynamic programming fashion to update the minimum routing cost of its associated resource node iteratively until the minimum routing cost of every node keeps stable. Note that a node is stable when its routing cost

will not change. At last, the array A is copied from off-chip DRAM into main memory.

Optimization Approach. In a GPU architecture, accessing the data of on-chip shared memory is much faster than accessing the data of off-chip DRAM. We optimize the fine-grained parallel routing approach by exploiting on-chip shared memory. Since the data size of on-chip shared memory is small and limited in each SM, it poses a challenge to import all of the routing data into the shared memory when increasing design scale.

To address this problem, we partition the routing box into several equal-sized sub-boxes such that the data size of each sub-box is smaller than the capacity of shared memory of each SM, enabling the data of each sub-box can be imported into the shared memory of each SM. In this way, a single net routing in initial bounding box can be converted into two phases: one is that path searching is performed inside each sub-box and the other is that path searching is performed at the boundary nodes provided by the adjacent sub-boxes. Note that the adjacent sub-boxes mean that there exist at least one edge connecting two nodes that are distributed to the boundaries of two different sub-boxes.

```
Given: 1) A set of boundary nodes from each sub-box, B.
       2) A sub-box, Si.

Begin
  The net box is partitioned into several sub-boxes Si.
  Each sub-box Si is imported into shared memory of its own SM.
  While (An unstable node exists) {
      ParaBegin
          // Updating the costs of boundary nodes B.
          A[n, k] = min{A[m, k-1] + c(m, n), A[n, k-1]}.
      ParaEnd
      ParaBegin
          // Updating the costs of nodes inside a sub-box Si.
          While (An unstable node exists inside a sub-box Si) {
              A[n, k] = min{A[m, k-1] + c(m, n), A[n, k-1]}.
          }
          End While
      ParaEnd
  }
  End While
End
```

Fig. 10. A key sketch of optimization approach.

Given a single net, we have its bounding box and know its routing resource nodes, as well as the connections of each node. When the net bounding box is partitioned into several equal-sized sub-boxes, we are able to classify these nodes into two types: one is the nodes in the interior of a sub-box and the other is the nodes on the boundary of each sub-box. The nodes located in the sub-box interior can be imported into the shared memory of its own SM. Fig. 10 gives the key sketch of dynamic programming based optimization approach and in the parallel routing of a single net, we still adopt the same dynamic programming algorithm to update the routing cost of every resource node alternately. Specially, updating the costs of boundary nodes between adjacent sub-boxes is prior to the updating in each sub-box.

In this optimized Scheme-2 approach, the details of updating the costs of nodes in a sub-box are as follows. Note that the path searching of sub-box enables each sub-box as an independent routing region and each GPU thread has its own memory space in on-chip shared memory that is used to store the costs of the nodes. Initially, each GPU thread is responsible for its own sub-box, copying cost array from off-chip DRAM into its own on-chip shared memory. And then, each GPU thread leverages the same algorithm to update the costs of nodes in the shared memory until all of the nodes inside the sub-box become stable. At last, the stable cost array is copied back into the off-chip DRAM.

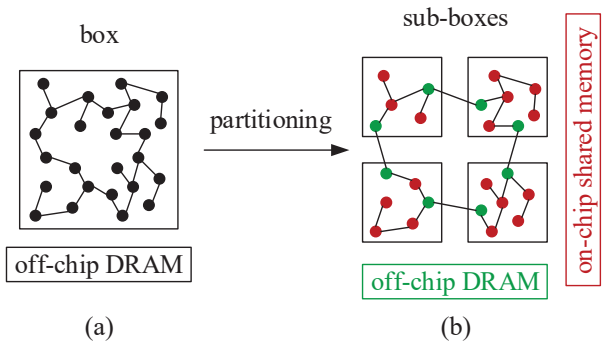

Fig. 11. An example of Optimized Scheme-2 approach. The red nodes are inside the sub-boxes and the green nodes are on the boundary of each sub-box.

To demonstrate the updating of boundary nodes, we give an example of partitioning a net box into four sub-boxes as shown in Fig. 11. At the end of partitioning, two phases are iteratively performed to update the costs of all the resource nodes that belong to the net box. The path searching of adjacent sub-boxes updates the costs of the green nodes on the boundary of each sub-box and changes their cost statuses. And then, the path searching in a sub-box updates the costs of the red nodes and implements local stability of their cost statuses. As given in Fig. 10, for each sub-box, these two alternating phases iterate until the costs of all nodes in a sub-box become stable. Note that the costs of the boundary nodes between adjacent sub-boxes are updated through off-chip DRAM. The costs of each sub-box are updated in on-chip shared memory.

By partitioning the net bounding box into several small sub-boxes, we have the ability to exploit the on-chip shared memory of the GPU to accelerate the routing. With this optimization approach, our Scheme-2 can obtain significant speedup.

V. Evaluation

A. Experimental Setup

Experiments are performed on a platform with a Nvidia GeForce GPU and an Intel Xeon E5-2650 CPU. The GPU has 15 SMs (1920 cores) and 8GB global memory (off-chip

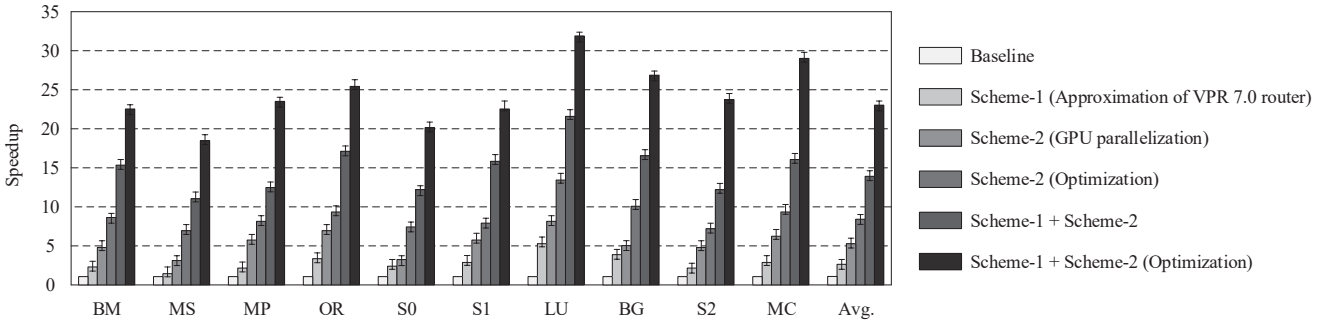

Fig. 12. The achieved speedups in our proposed parallel router. Baseline is the original VPR 7.0 router. Scheme-1 is the approximation of VPR 7.0 router. Scheme-2 is the parallelization of VPR 7.0 router on the off-chip DRAM of the GPU. Scheme-2 (Optimization) is parallelization of VPR 7.0 router on the on-chip shared memory of the GPU. Scheme-1 + Scheme-2 denotes the parallelization of the Scheme-1 router on the off-chip DRAM of the GPU. Scheme-1 + Scheme-2 (Optimization) denotes the parallelization of the Scheme-1 router on the on-chip shared memory of the GPU.

DRAM). The CPU has 2.3 GHz clock frequency and 64GB memory. The data communication between the host CPU and the device GPU adopts the PCI Express 3.0 lanes, operating at 16x speed.

TABLE II
BENCHMARK SUMMARY

Name	Abbr.	Arch.	Size	Nets	CLBs
blob_m.	BM	k4_N4_90nm	51x51	6606	2702
mkSM.	MS	k4_N4_90nm	53x53	7154	3126
mkPK.	MP	k4_N4_90nm	58x58	7474	3767
or1200	OR	k4_N4_90nm	65x65	8078	3648
stere.0	S0	k6_N10_40nm	39x39	9312	1492
stere.1	S1	k6_N10_40nm	39x39	13523	1401
LU8PE.	LU	k6_N10_40nm	53x53	16278	2373
bgm	BG	k6_N10_40nm	73x73	27853	4225
stere.2	S2	k6_N10_40nm	86x86	36479	2802
mcml	MC	k6_N10_40nm	101x101	81282	7934

The proposed parallel router is evaluated with the 10 largest benchmarks in Table II from the VTR 7.0 benchmark suite commonly used in FPGA CAD research [21]. We use ABC for logic synthesis and technology mapping and use T-VPack and VPR placer for packing and placement. Across all runs every benchmark was routed using a channel width of $1.4\times$ the minimum channel width needed by VPR router, where the used channel width is same with most of the previous works on parallel FPGA routing. Moreover, the VPR tool consists of routability- and timing-driven routers, both of which are based on PathFinder algorithm, having same data structure and algorithmic flow. In this paper, we only parallelize the routability-driven router to evaluate the achieved speedup and total routed wirelength. Note that all of the experiments are compared with the original VPR 7.0 routing algorithm which forms our baseline.

B. Speedup Analysis

Fig. 12 shows all of the achieved speedups using Scheme-1 and Scheme-2 in parallel routing. The first bar of each cluster is the results of baseline. In each cluster, the second bar is the results of using Scheme-1 only, the third is the results of using Scheme-2 only, the fourth is the results of using optimized Scheme-2 only, the fifth is the results of using Scheme-1 and Scheme-2 together, and the sixth is the results of using Scheme-1 and optimized Scheme-2 together. We evaluate the impacts of each approach on speedup of our parallel router one by one as follows.

We first evaluate the impact of the Scheme-1 on the runtime of the serial VPR 7.0 router. The original serial router adopts the default bounding box expansion and it expands the net box in four directions. With only Scheme-1, the original serial router can be approximated by using the congestion-aware selective expansion. On average, the approximated serial router can provide about $2.83\times$ speedup comparing to original serial router. This advantage is due to that the approximated serial router enables the single net to be routed in its own bounding box and only expands the net routing box in a single direction. Note that both original and approximated serial routers are performed on the CPU platform. Moreover, the achieved speedup is the ratio of routing time between the original serial router and the approximated serial router.

We then evaluate the impact of the Scheme-2 on the speedup of the serial VPR 7.0 router. When using Scheme-2 only, the serial router can be accelerated on a GPU platform. On average, the speedup of about $5.17\times$ can be achieved by the GPU-accelerated router when comparing to the original serial router. Note that the achieved speedup is the proportion of the runtime of the original serial router and the runtime of the GPU-accelerated router. The original serial router and the GPU-accelerated router are performed on the CPU and GPU platforms, respectively. Moreover, the Scheme-2 enables the GPU-accelerated router to leverage the off-chip DRAM of the GPU to accelerate the routing time.

To exploit the on-chip shared memory of the GPU, we optimize the Scheme-2 to obtain better speedup on the GPU platform. In optimized Scheme-2, we partition the net bounding box into several sub-boxes, each of which is performed at the on-chip shared memory of the GPU. On average, the optimized Scheme-2 approach enables the GPU-accelerated router to provide about $8.52\times$ speedup, comparing to the original serial router. This optimized Scheme-2 has about $1.65\times$ faster than the initial Scheme-2. This improvement is due to that the optimized Scheme-2 enables the GPU-accelerated router to leverage the on-chip shared memory of the GPU. Note that processing the data of on-chip shared memory is much

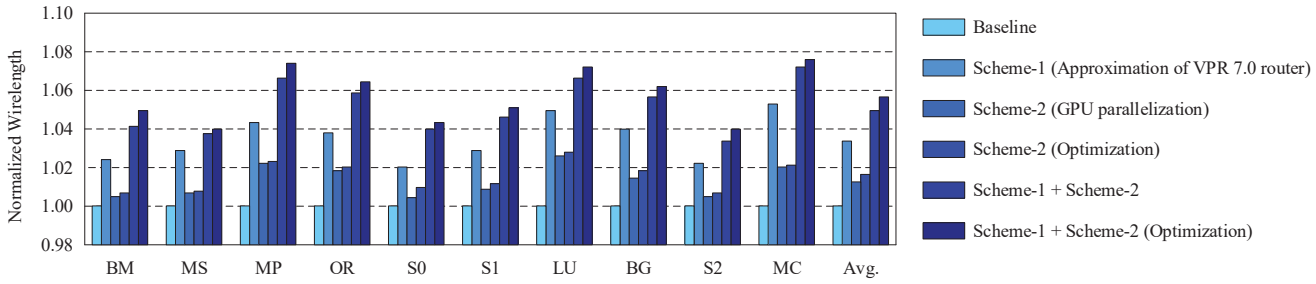

Fig. 13. The normalized total routed wirelength in our proposed parallel router. Baseline is the original VPR 7.0 router. Scheme-1 is the approximation of VPR 7.0 router. Scheme-2 is the parallelization of VPR 7.0 router on the off-chip DRAM of the GPU. Scheme-2 (Optimization) is parallelization of VPR 7.0 router on the on-chip shared memory of the GPU. Scheme-1 + Scheme-2 denotes the parallelization of the Scheme-1 router on the off-chip DRAM of the GPU. Scheme-1 + Scheme-2 (Optimization) denotes the parallelization of the Scheme-1 router on the on-chip shared memory of the GPU.

faster than processing the data of off-chip DRAM on a GPU platform. Also, the optimized GPU-accelerated router and the original serial router are performed on the GPU and CPU platforms, respectively. The achieved speedup is the time of the original serial router divided by the time of the optimized GPU-accelerated router.

At last, we put Scheme-1 and Scheme-2 together and integrate them into the serial VPR 7.0 router, further evaluating the total speedup of the fine-grained parallel router. The baseline is still the original serial router that uses the default expansion technique, running on the CPU platform. In this basic fine-grained parallel router, Scheme-1 enables each net to use the congestion-aware expansion technique on the CPU platform and Scheme-2 enables each net to be parallel routed on the off-chip DRAM of the GPU platform, both of which make the fine-grained parallel router provide about 14.06× speedup on average, comparing to baseline. Note that the achieved speedup is the ratio between the CPU runtime of baseline and the CPU+GPU runtime of parallel router.

To improve the total speedup of the basic fine-grained parallel router, we impose the proposed optimization approach to Scheme-2. With Scheme-1 and optimized Scheme-2, the basic parallel router can be optimized to provide about 23.75× on average, comparing to the original serial router. This optimized parallel router has about 1.7× improvement over the basic parallel router. This improvement is due to that the optimized parallel router enables each net to be parallel routed at the on-chip shared memory rather than the off-chip DRAM on the GPU platform. Again, the achieved speedup is the ratio between the CPU runtime of original serial router and the CPU+GPU runtime of optimized parallel router.

C. Total Routed Wirelength

We evaluate the impacts on total routed wirelength when using our Scheme-1 and Scheme-2 approaches. We only employ Scheme-1 and it expands the routing box of single net in a single direction. Although this scheme limits path space to reduce routing time, the net will be forced to detour in its own bounding box. On average, Scheme-1 results in about 4% longer total routed wirelength compared to the original VPR 7.0 router. Then we evaluate the impact of Scheme-2 on total routed wirelength. With only Scheme-2, the path searching space of the single net is the default routing resource region. Thus the degradation of wirelength is relatively small when parallelizing the single-net routing on a GPU platform. When combining Scheme-1 and optimized Scheme-2 together, our parallel router results in about 6% bigger minimum number of wires required for routing on average than serial VPR 7.0 router. Fig. 13 shows the normalized total routed wirelength in our proposed parallel router.

D. Comparisons with Recent Works

We compare the proposed parallel routing approach with other two parallel routing approaches, Bamboo [17] and ParaDRo [18], regarding the total speedup and quality of results. Note that Bamboo is the state-of-the-art GPU-based parallel router and ParaDRo is the state-of-the-art CPU-based parallel router. Here we do not emphasize the differences such as the number of benchmarks, the experimental platforms and so on, but try to provide general observations based on the proposed approach and the total achieved speedup. This comparison policy is the same as the previous parallel routing works, including Bamboo and ParaDRo.

In terms of total speedup, ParaDRo can provide an average speedup of 5.4× using 8 processor cores on the CPU platform while Bamboo can provide 15.13× speedup on average using 2880 cores on a Tesla K40c GPU with 12GB memory. Note that both ParaDRo and Bamboo are also compared with the serial VPR 7.0 router. In our proposed approaches, the Scheme-1 provides a close approximation of the serial router and achieves on average 2.83× speedup only on the CPU platform. Note that this Scheme-1 is a runtime reduction approach and it does not have any parallelization overheads. The optimized Scheme-2 enables the serial router to be parallelized only on the GPU platform and it is able to provide an average speedup of 8.52× using 1920 GPU cores.

When putting Scheme-1 and optimized Scheme-2 together and integrating them into the original VPR router, the total speedup of 23.75× on average can be achieved on the CPU and GPU platform. Benefiting from the combination of the approximation of VPR 7.0 router and its parallelization, our approach is faster than Bamboo and there has a 1.57× speedup improvement. When comparing with the maximal speedup of ParaDRo, our approach can implement a 4.39× improvement

in total speedup. Fig. 14 shows the relatively comparisons between our proposed approaches and other two state-of-the-art works. Note that the total speedup is an average value, which can be obtained from their experiments. These comparison results denote that combining serial approximation and GPU parallelization has the potential to obtain significant speedup.

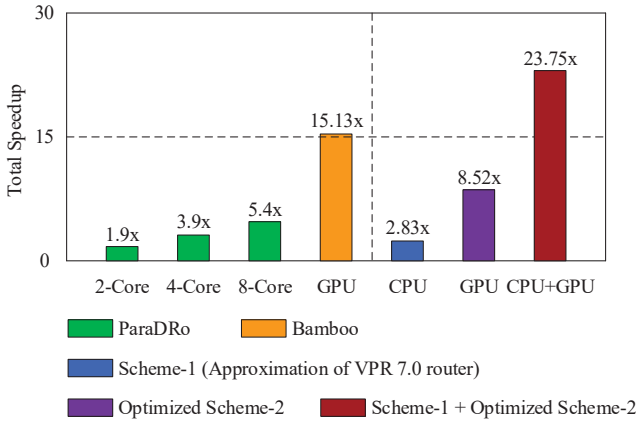

Fig. 14. Comparisons with recent parallel routers. Scheme-1 is the approximation of VPR 7.0 router. Optimized Scheme-2 is parallelization of VPR 7.0 router on the on-chip shared memory of the GPU. Scheme-1 + Optimized Scheme-2 denotes the parallelization of the Scheme-1 router on the on-chip shared memory of the GPU.

In terms of quality, on average, Bamboo has about 1% degradation on a GPU platform [17] and ParaDRo leads to about 5% degradation with 8 processing cores [18], both of which are compared with the original serial VPR 7.0 router. Our parallel router is about 6% degradation and this is slightly higher than these two parallel routers. The previous work [19] also studies that how much quality degradation is necessary to accelerate the routing time. Our parallel router trades about 6% degradation in total routed wirelength for about 24× speedup. It is acceptable to employ our approach to accelerate the routing for FPGAs.

VI. Conclusion

In this paper, we propose a fine-grained GPU-based parallel routing approach that exploits two new schemes for significant speedup. The Scheme-1 is a runtime reduction approach and in this approach, we utilize the iterative feature of routing algorithm and the congestions between nets to expand the routing box of each net in a single direction. The Scheme-2 is a GPU parallelization approach and in this approach, we regard each node as a GPU thread and perform path searching in dynamic programming algorithm on all the nodes simultaneously. Moreover, the Scheme-2 can be optimized by partitioning the routing box of single net into several sub-boxes, each of which can be processed at the on-chip shared memory of the GPU. When putting Scheme-1 and optimized Scheme-2 together, our approach of combining approximated router and its parallelization provides total speedup of about 24× with a 6% wirelength degradation on average, over the original VPR 7.0 router.

VII. Acknowledgement

We appreciate the insightful comments and feedbacks from anonymous reviewers. This work was supported by 2018YFB1003502, NSFC61433019, NSFC61802446, and 2016ZT06D211.

References

[1] Altera. *Stratix 10 product table.* 2016.
[2] Xilinx. *UltraScale architecture and product overview.* 2016.
[3] H. Esmaeilzadeh et al. *Dark silicon and the end of multicore scaling.* In Proceedings of the International Symposium on Computer Architecture (ISCA), 2011.
[4] R. Prabhakar et al. *Plasticine: A reconfigurable architecture for parallel patterns.* In Proceedings of the International Symposium on Computer Architecture (ISCA), 2017.
[5] J. Ouyang et al. *Software-defined accelerator for large-scale DNN systems.* In Proceedings of the International Symposium on High Performance Chips (Hot Chips), 2014.
[6] J. Fowers et al. *A configurable cloud-scale DNN processor for real-time AI.* In Proceedings of the International Symposium on Computer Architecture (ISCA), 2018.
[7] S. Karandikar et al. *FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud.* In Proceedings of the International Symposium on Computer Architecture (ISCA), 2018.
[8] L. McMurchie and C. Ebeling. *Pathfinder: A negotiation-based performance-driven router for FPGAs.* In Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), 1995.
[9] R. Fung, V. Betz, and W. Chow. *Simultaneous short-path and long-path timing optimization for FPGAs.* In Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2004.
[10] S. Gupta, J. Anderson, L. Farragher, and Q. Wang. *CAD techniques for power optimization in virtex-5 FPGAs.* In Proceedings of the IEEE Custom Integrated Circuits Conference (CICC), 2007.
[11] M. Gort and J. Anderson. *Deterministic multi-core parallel routing for FPGAs.* In Proceedings of the International Conference on Field-Programmable Technology (FPT), 2010.
[12] Y. Moctar and P. Brisk. *Parallel FPGA routing based on the operator formulation.* In Proceedings of the annual Design Automation Conference (DAC), 2014.
[13] M. Shen and G. Luo. *Accelerate FPGA routing with parallel recursive partitioning.* In Proceedings of the International Conference on Computer Aided Design (ICCAD), 2015.
[14] M. Shen and G. Luo. *Corolla: GPU-accelerated FPGA routing based on subgraph dynamic expansion.* In Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), 2017.
[15] C. Hoo and A. Kumar *ParaDiMe: A distributed memory FPGA router based on speculative parallelism and path encoding.* In Proceedings of International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017.
[16] M. Shen, N. Xiao, and G. Luo. *A coordinated synchronous and asynchronous parallel routing approach for FPGAs.* In Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2017.
[17] M. Shen, N. Xiao, and G. Luo. *Dependency-aware parallel routing for large-scale FPGAs.* In Proceedings of the International Conference on Computer Design (ICCD), 2017.
[18] C. Hoo and A. Kumar. *ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling.* In Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), 2018.
[19] C. Mulpuri and S. Hauck. *Runtime and quality tradeoffs in FPGA placement and routing.* In Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), 2001.
[20] K. Murray et al. *Timing-Driven Titan: Enabling large benchmarks and exploring the gap between academic and commercial CAD.* In Proceedings of ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2015.
[21] J. Luu et al. *VTR 7.0: Next generation architecture and CAD system for FPGAs.* In Proceedings of ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2014.