POLITECNICO DI TORINO
Repository ISTITUZIONALE

Routino: Accelerating FPGA Routing through Efficient Memory Representation

Original
Routino: Accelerating FPGA Routing through Efficient Memory Representation / Nicolini, D., De Sio, C., Vacca, E., Sterpone, L.. - ELETTRONICO. - (2025), pp. 168-176. (35th International Conference on Field-Programmable Logic and Applications Leiden (NL) 1 - 5 September 2025) [10.1109/FPL68686.2025.00033].

Availability:
This version is available at: 11583/3002669 since: 2025-09-01T11:31:27Z

Publisher:
IEEE

Published
DOI:10.1109/FPL68686.2025.00033

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

12 October 2026

Routino: Accelerating FPGA Routing through Efficient Memory Representation

Davide Nicolini, Corrado De Sio, Eleonora Vacca, Luca Sterpone
Politecnico di Torino, Turin, Italy

{davide.nicolini, corrado.desio, eleonora.vacca, luca.sterpone}@polito.it

***Abstract*— The rapid increase in the complexity of Field-Programmable Gate Arrays (FPGAs) is significantly impacting the efficiency of the design implementation flow. In particular, the routing process presents challenges in achieving computational efficiency and reducing time-to-solution due to the increasing on-chip resources and device complexity. This work introduces a novel router that leverages optimized data structures and memory access patterns to minimize memory consumption. Experimental results prove how the proposed approach can significantly reduce time-to-solution, identifying memory consumption as a barrier to achieving scalability and proposing solutions based on FPGA modular architecture to face it, achieving an average memory usage reduction of about 90% and an average decrease of routing time of 40%.**

Keywords—FPGA, Routing, Memory Management, Programmable Logic

I. INTRODUCTION

In the last few years, Field-programmable Gate Array (FPGA) devices have transitioned from architectures based on a limited number of elementary and low-performance logic blocks to complex heterogeneous architectures. New architectures include sophisticated macro resources such as block RAMs (BRAMs) and Digital Signal Processing (DSP) units, Look-Up-Table (LUT)-based logic resources, and complex routing infrastructures. However, as the complexity and number of available resources increase, the routing process—already a significant concern in programmable logic and ASIC design—becomes even more challenging. Indeed, in the FPGA design flow, the issue of logic synthesis for complex design does not escalate as drastically as routing because it mainly depends on the FPGA available resources and the designer's skill in clearly and efficiently specifying the intended circuit behavior. Conversely, the routing process must address multiple challenges induced by this growth in complexity, requiring an increasing number of optimizations to achieve fast routing time and good memory management, which are usually delegated to vendor-specific tools. Such tool releases are becoming increasingly demanding regarding memory to support the latest devices. For instance, AMD states that the current release of Vivado can require over 60 GB of RAM (and much over 100 GB of permanent memory) when dealing with the latest FPGA family, resulting in a significantly extended processing time, even for relatively small designs. FPGAs continue to increase in complexity and performance, but the progress of these tools in addressing this bottleneck remains marginal. Consequently, the utilization of dedicated computers has frequently become necessary to fully leverage the capabilities of the latest FPGA architectures, compromising an advantage of these systems originally conceived as cost-effective solutions for custom hardware design and rapid prototyping.

As technology scaling approaches the sub-nanometer size, dense and complex routing infrastructures characterize new FPGA architectures. The increase in on-chip resources for routing results in a broader solution space to explore for which traditional implementations of routing algorithms are not enough to address the problem efficiently. Well-established algorithms such as Maze [1], A*[2], and Pathfinder [3] have been widely used in both literature and industry due to their proficiency in handling routing tasks. For decades, efforts have focused on optimizing these algorithms to reduce congestion and minimize critical path lengths. However, the industry's primary focus is moving toward the time-consuming nature of implementation of such approaches, particularly as the number of available resources and design complexity continue to increase. Such a problem is becoming so impending that AMD proposed a worldwide routing contest, where achieving a fast-routing solution was nine times more important than minimizing critical paths [4]. Given the proprietary nature of the vendor's tools, it is complex to pinpoint the exact bottlenecks that cause a substantial delay in achieving solutions during the routing process. Nevertheless, profiling memory usage in AMD Vivado and inspections of open-source routers such as RWroute [5] and VPR [6] suggest that one potential issue might stem from how routing resources are represented.

Commonly, routing algorithms heavily rely on a representation of the routing resources available in the device, named Routing Resource Graph (RRG). The RRG embeds numerous vertices and edges, increasing as the architecture complexity of the device increases. We identified the overhead of managing an in-memory representation RRG as one of the primary candidates for slowing down the routing process and increasing the computational complexity. Due to memory constraints, a common solution in routing tools is to load only a subsection of the nodes of RRG in memory to satisfy the design constraint and congestion. However, due to the iterative approach for routing designs, additional elements of RRG must often be fetched and continuously added, iteration after iteration, to the memory representation of RRG to expand the search space to achieve a compliant solution. As a result, the size of the memory representation of RRG increases exponentially with each iteration, often saturating system resources and significantly impacting performance. This insight highlights an area where further investigation may lead to the optimization of routing efficiency relying on a lightweight representation of the RRG.

In this work, we propose a methodology and present a router, named Routino, for optimizing the representation of the FPGA routing architecture for minimal memory allocation based on the high regularity of FPGA systems. As a result,

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

Routino achieves a memory usage reduction by a factor ranging from 5.79 to 26.68 and decreases routing time by a factor ranging from 1.52 to 41.86 times compared to AMD Vivado. Compared to RWroute, it reduces memory usage by a factor ranging from 6.9 to 37.93 times and shortens routing time up to 11.63 times. On average, Routino presents a memory consumption reduction of 90% and a decrease of Routing time of 40%. The routing quality, in terms of used routing resources and critical path delay, is about 20% higher for small designs but is negligible for medium and large designs. Of primary interest is that these improvements are mainly achieved by efficiently representing the available resources in the FPGA device, an approach that other, higher-performance routers can also take advantage of.

II. Related works

Research efforts to optimize FPGA CAD tools predominantly focus on improving runtime and performance. State-of-the-art solutions predominantly aim to accelerate the routing process by employing either dedicated hardware accelerators, such as GPUs[7], or multi-threading techniques[8] that distribute routing tasks across up to 32 CPU cores[9]. While these methods significantly reduce routing time, they often require substantial computational resources, sometimes exceeding the actual demands of the routing problem itself. An alternative and largely unexplored approach to routing time optimization lies in efficient memory management. Specifically, the representation of the RGG as a more compact data structure offers a dual advantage: it not only reduces memory consumption but also may enhance routing time efficiency. By refining the way routing data structures are stored and accessed, this approach has the potential to achieve routing time improvements without necessitating excessive computational overhead. Pioneer work in this scenario is that of Chin and Wilton[10], which is the only notable study addressing memory usage in FPGA routing algorithms. Their technique focuses on reducing redundancy in the representation of routing resources by leveraging the structural similarity of tiles in the FPGA architecture, considering that the FPGA fabric consists of identical or similar blocks (tiles) arranged in a grid-like structure. By representing each tile type with a single graph and using tile coordinates to determine edge validity, they minimize memory requirements. This approach achieves a significant memory reduction of 5–13×, albeit with a modest runtime overhead of 2.26× for routing and 1.28× overall. Since their work is based on VPR, they model the entire FPGA using only five tile types. However, this approach is overly simplistic for real-world scenarios, as it assumes a high degree of uniformity and regularity in FPGA architectures that do not reflect modern commercial FPGAs. Commercial FPGAs, such as the Xilinx UltraScale+ family, feature a far more intricate tile-based structure with numerous tile types and diverse connection patterns, including specialized blocks like DSPs, BRAMs, clock management tiles, and high-speed I/O tiles. Representing such diversity with a small number of tile types would lead to inaccuracies or loss of critical detail, making the method impractical for commercial designs. Furthermore, the reliance on coordinate-based edge validation adds computational overhead for the router, as it must repeatedly compute edge validity based on tile coordinates, which becomes less efficient and scalable for the larger and more heterogeneous architectures of commercial FPGAs.

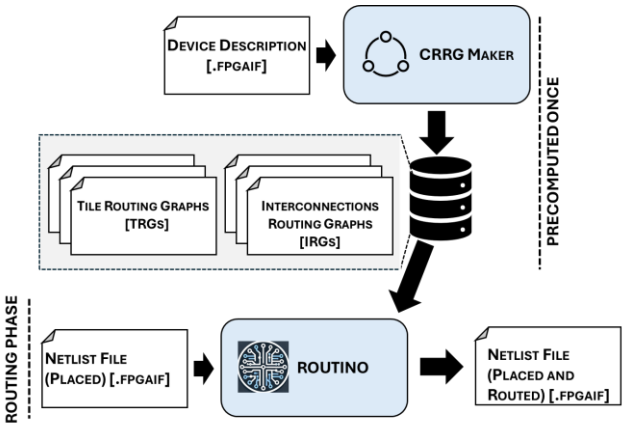

Fig. 1. Conceptual Schema of Proposed Router and Methodology.

In contrast, our approach targeting commercial FPGA addresses these complexities by assigning each tile a connection template that can be reused by other tiles with similar connection patterns. While this method requires additional memory allocation with respect to [10], it eliminates the need for the router to verify coordinates for edge validity. By doing so, our proposed solution eliminates the runtime overhead faced by [10] while maintaining the memory management efficiency, providing as good tradeoff between memory consumption and routing time as well as scalable solution for commercial FPGA architectures.

III. Routino's Resource Representation and Routing

A. Overview

We propose a new router called Routino, developed in C++, based on the efficient representation of RRG. For clarity, the current section uses terminology and concepts based on AMD UltraScale+ devices. However, the methodology easily applies to virtually any vendor FPGA architecture due to the high redundancy of components that typically characterize such devices. A primary feature of Routino is to ensure a low memory footprint by reducing RRG's memory usage.

Fig. 1 shows Routino's conceptual schema. The device architecture described using the FPGA Interchange Format (FPGAIF) [11], a standard that addresses interoperability issues caused by proprietary formats, is the starting point of the methodology. Used by routers like RWroute, FPGAIF ensures seamless integration with vendor tools despite the growing complexity of routing resources. From this description, a condensed representation of RRG (CRRG) is generated by creating multiple subgraphs that exploit the inherent redundancy within the FPGA architecture. Even if this task is computationally demanding it is needed once per device and can be easily precomputed. The result of this stage is what enables a lightweight representation of the device architecture, which also results later in a speed-up in finding the routing solution. Routino uses the generated CRRG for routing placed netlists during the routing step, following the traditional FPGA design development flow. Placed netlists,

as well routed netlists obtained as the output of the routing process, are provided using FPGAIF format.

B. Routing Infrastructure

The latest FPGA family architectures are characterized by hundreds of thousands of tiles. A tile is a basic building block of the FPGA architecture. These blocks are replicated across the FPGA, serving different functions, embedding sequential and combinational logic, programmable interconnections, fixed interconnections, or even empty spaces. Despite the large number of tiles within a modern FPGA, the variety of tile types remains relatively limited. For instance, in the Xilinx UltraScale+ family, there are approximately 200 distinct tile types, yet the entire FPGA structure is composed of a vast number of individual tiles, 208,370 in this case.

The goal of the router is to obtain a valid routing solution, which means finding an efficient routing path from the starting to ending points of each net using the routing resources available on the FPGA without conflicts. To achieve that, routers rely on the RRG to describe and model the elements of the specific FPGA architecture.

In the context of AMD's FPGAs, reported in Fig. 2, the nodes of the device represent the vertexes of the RRG, and the edges by the *Programmable Interconnection Points* (PIPs). PIPs are the only programmable routing lines, and so they are, with a few exceptions, the only element on which we can act to control routing. A **node** is a fixed routing line spanning over tiles that connects different tiles. A node is composed of one or more *wires*. A *wire* is a segment of a node

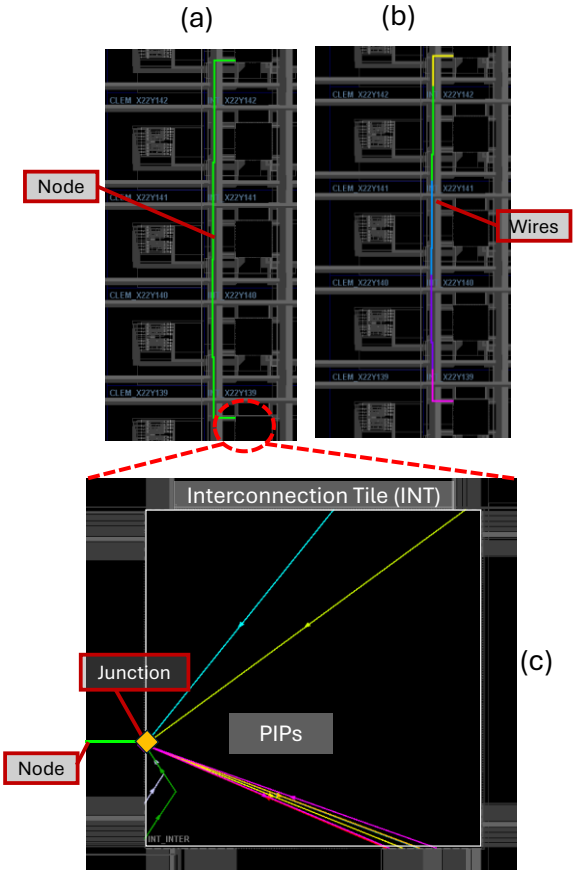

Fig. 2. Overview of the element of the routing structure. (a) node spawning over multiple tiles; (b) wires ; (c) interconnection tile structure

in a specific tile. So, if a node spans through 4 tiles, there will be 4 wires, one for each tile. Although not explicitly represented in the RRG, *pins* play a crucial role in the routing architecture. A **pin** is a logic resource's connectivity point. A net connects pins, starting with a single pin and ending with one or more pins. To route a circuit means finding a path, using nodes and pips, that connects the starting pin (source) with the end pins (sink) of all the nets. A **junction** is the point of connectivity of the **Interconnection Tiles (INT),** differing from pins since they cannot be drivers or sinks. An INT tile includes thousands of programmable routing resources called **PIPs** that allow the connection of two junctions and, consequently, of two nodes, thus creating a connection between two pins or junctions.

C. Condensed Routing Resource Graph

RRG represents the FPGA's routing infrastructure, which will be traversed to find routing paths for all the nets in the design. A routing path is defined univocally by activating PIPs. Indeed, since the location of the driver and sinks of a net are fixed during the placement phase, they are bound to a specific vertex in the graph. The goal is to find a set of PIPs that enable a path connecting the driver's node with the sinks' nodes (avoiding conflicts). Since PIP to make active is the only choice when we build a routing path, nodes are modeled as vertices and PIPs as edges in the RRG. To clarify the size of this graph, an XCVU3P, the smallest device of the Virtex UltraScale+ series, has 28,226,432 vertices and 130,147,269 edges. The size of the RRG significantly impacts memory requirements and computational performance during the routing process. To mitigate this issue, we introduce the concept of Condensed Routing Resource Graph (CRRG) which takes advantage of the inherent regularity and repetitive patterns found in FPGA architectures to reduce the memory footprint without compromising the essential information required for effective routing. Instead of maintaining a separate representation in the graph for every individual node and PIP, the CRRG combines these elements. It achieves this by representing similar tiles only once, thereby avoiding redundant data storage. This method ensures a more compact and efficient representation of the routing resources while retaining the necessary detail for accurate and high-performance routing. This significantly reduces redundancy and allows for a more compact representation. The CRRG consists of two different sets of graphs, a set of Tile Routing Graphs (TRGs) and a set of Interconnection Routing Graphs (IRGs).

D. Tile Routing Graphs (TRGs)

A TRG models how a tile is connected to its neighbor's tile via nodes. An IRG models the internal representation of INT Tiles, particularly regarding PIPs and nodes. The TRGs leverage repetitive patterns in tile positioning within the FPGA. In particular, each tile is classified into a template. All the tiles with identical relative neighbors belong to the same template. A TRG encodes a template. In detail, a TRG is structured as a star graph, where a central vertex, named the core tile, is connected to the surrounding vertices, called neighboring tiles. The neighboring tiles are those that, in the FPGA fabric, can be reached from the core tile using a single node. Please note that even if a neighbor is reachable using a single node, the node can span over multiple tiles horizontally

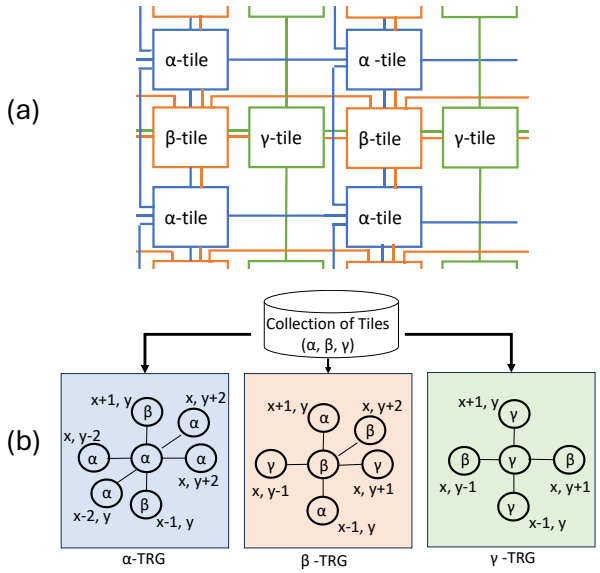

Fig. 3. Conceptual schema of TRGs (b) derived from the FPGA architecture (a).

or vertically. As a result, a TRG models the neighbors of all the tiles of a given template and includes information for each neighbor on which node is used for reaching it, as well as the relative position and the template of the neighbor itself. Due to repetitive patterns in FPGA architectures, many tiles can share the same template. We want to emphasize that the TRG embeds information agnostic from the real position of the core tile, relying on relative names and positions with respect to the core tile. For clarity an example is presented in Fig. 3. Suppose two different templates, α and β exist in an FPGA architecture associated respectively with α-TRG and β-TRG. The α-TRG can be exploited to find that the α-tile with coordinates (X, Y) has a neighbor that is a β-tile at coordinates (X+1, Y) reachable using a node (whose name is embedded in the TRG, e.g., X_Y/SS1_W_BEG5). At this point, TRG_β must be used to explore neighbors of the β-tile located at (X+1, Y). The β-tile has a neighbor that is an α-tile at (X+1, Y). If the starting α-tile had coordinates (20, 50), the β-tile would be located at (21,50) and the second α-tile at (22,50). The latter α-tile shares the α-TRG with the former

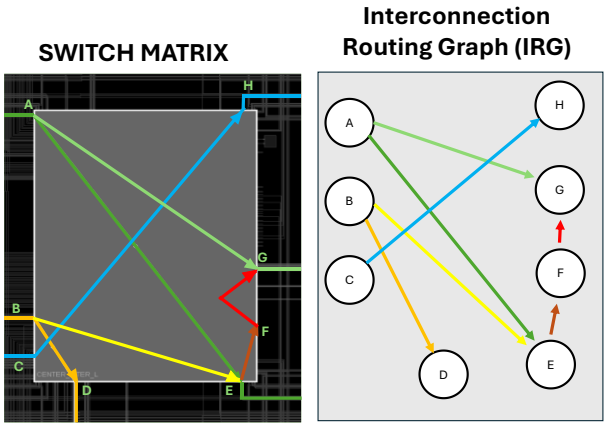

Fig. 4. Conceptual schema of IRGs deriving from a simplified version of a switch matrix.

since both belong to α-template, so it will have a neighbor that is a β-tile at coordinates (X+1, Y), i.e. (23,55), reachable using node 26_53/SS1_W_BEG5 as well. For instance, using this methodology, we modeled information of ~100,000 tiles used in the routing procedure of a Virtex XCVU3P, the smallest UltraScale+ part, in ~1,500 entries while still providing a full representation of the FPGA routing resources and architecture in memory.

E. Interconnection Routing Graphs (IRGs)

The IRGs are a set of graphs used to model the routing structure of the interconnection tiles. Specifically, the vertices represent junctions or nodes (since each junction is one-to-one associated with a routing node) within the tile, while the edges represent the PIPs. The IRGs are used to identify a path between sets of junctions within the INT tile, i.e. a junction connected to the input node and one (or more) junction connected to an output node. Fig. 4 reports an example of an IRG extracted by a simplified switch matrix (or interconnection tile). Please note that real INT tiles are characterized by thousands of junctions and more than 4,000 PIPs for the latest devices. Moreover, a junction can have both incoming and outgoing PIPs. Additionally, some junctions, named bounce junctions, are not associated with routing nodes leaving the tile. Their purpose is to be used for connecting two junctions in the INT tile that do not have a direct PIP connecting them but can be connected using a pair (or more) of PIPs. As a result, an IRG presents a non-trivial structure and it is generally more complex than TRGs.

F. Use of Routing Resources

Building the TRGs and IRGs and associating the tiles with the correct template for each FPGA device is an operation performed once offline, hence not affecting the routing time. Meanwhile, sharing representations across different tiles significantly reduces memory usage. However, in a traditional RRG, each routing resource is uniquely represented, allowing for resource-specific attributes such as cost and usage to be directly associated with individual graph elements. This explicit one-to-one mapping ensures that detailed, resource-specific data can be stored and updated without ambiguity. In the CRRG, multiple resources are combined into a single representation to leverage the regularity and repetitive patterns of FPGA architectures. While this reduces memory usage significantly, it introduces a challenge: because a single graph element in the CRRG corresponds to multiple physical resources, associating unique attributes like cost or usage with individual resources becomes infeasible. A mapping strategy is employed to address this, where each wire is assigned a unique identifier. This identifier serves as a key to access the corresponding resource-specific data stored in an external map. This method preserves the ability to manage resource-specific attributes efficiently, despite the condensed representation in the CRRG.

To manage resource-specific attributes efficiently in the CRRG, the map employs a lazy initialization strategy with default values. Each resource is initialized with a default value for its associated data, eliminating the need for immediate allocation of the full map. Allocation within the map occurs only when the resource's data deviates from this default value, ensuring that memory is used efficiently. This approach minimizes the initial memory footprint, as resources that retain their default values throughout the routing process do not consume additional memory. However, as routing progress and modifications are made to the data of individual

resources, the map dynamically grows to accommodate these changes. Consequently, while the CRRG itself remains compact, the map becomes the primary contributor to the program's memory usage, scaling dynamically as the graph is explored and resource data is altered.

G. Routing procedure

Routino is based on the Pathfinder combined with A*, but the CRRG allows for the integration of any routing algorithm. By using Pathfinder, the routing solution is found in multiple iterations. In each iteration, Routino attempts to route all the unrouted nets, net by net, while minimizing the cost function. If the iteration fails to find a conflict-free solution, nets presenting conflicts are unrouted and the cost of highly contested routing resources is increased. At this point, a new iteration begins attempting to route the unrouted nets using the updated costs. The adaptive cost adjustment prioritizes less congested resources in subsequent iterations, increasing the probability of convergence to a valid solution. The routing algorithm relies on A*, a depth-first search algorithm that uses a priority queue to explore tiles closest to the destination first.

The routing algorithm operates net by net, and each net explores the possible routing path routing node by routing node. Starting from the source node, the algorithm establishes a path to the destination. At each step, all vertexes (i.e., tiles) reachable from the current vertex are retrieved from the TRG and inserted into a priority queue. The priority of a vertex is computed using a heuristic that considers two primary factors: the Manhattan distance to the tile, favoring vertexes closer to the target, and the accumulated cost to reach that edge (i.e., the routing node) that penalizes highly contested resources due to congestion. The algorithm explores the graph always expanding the lowest-cost node first thanks to the priority queue. Thanks to that the net will select nodes that approach the destination. Upon reaching the destination, the path is reconstructed by backtracking from the destination node to the source. The sequence of used nodes is analyzed, and the least-cost path is selected for the final routing. The iterative nature of the approach, combined with congestion-adaptive cost adjustments, ensures efficient resource utilization. Thanks to TRG, it is trivial to find neighbor nodes when no INT tiles are involved since such tiles have often fixed routing only. Differently, the INT tiles present additional complexity. Indeed, for the INT tile, it is necessary to connect the input node to an output node using PIPs. To find a solution to such a sub-problem Routino relies on the IRG template associated with the specific INT tile. By identifying PIPs to be activated to connect the nodes, it is possible to reach the next tile. Since tiles may implement internal bouncing PIPs, this path may include multiple PIPs within the same INT tile. It is also important to highlight that, within IRG, A* is not applicable to find internal paths to connect with a node, because Manhattan Distance (i.e., the heuristic) cannot be applied within the same tile. However, in subsequent iterations, contested PIPs and junctions increase in costs, encouraging the use of other, less-utilized resources, balancing the load, and avoiding congestion.

H. Optimizations

Routino employs various optimizations to enhance both runtime performance and memory efficiency. In many cases, improvements in memory management have often resulted in better overall performance, likely due to reduced overhead from memory allocation. To achieve efficient memory usage, two primary strategies were adopted: lazy allocation for large data structures and a strict policy of retaining allocated memory unless it is guaranteed to be never used again. This approach ensures a consistently low memory footprint throughout the program's execution while also minimizing memory fluctuations due to continuous allocation and deallocation.

The router also benefits from optimizations designed to accelerate the routing procedure. One key optimization, that significantly speeds up routing, involves precomputing paths that connect multiple nodes without relevant branching, effectively collapsing them into single edges in the graph. Experimentally, we found these fixed paths to be especially common at the beginning and end of a route, specifically, when starting from the source node and when approaching the sink node. To further optimize routing, the feasibility of using these precomputed paths is evaluated before the main routing process begins, ensuring that their integration remains transparent to the core routing logic. Typically, these precomputed connections span only a few tiles, but in specific cases, they may extend over larger regions, significantly enhancing the overall efficiency of the routing process.

Typically, a routing algorithm iteratively routes and rips up each net, with nets usually being preserved if no conflicts are detected. Routino adopts this standard optimization but further refines it by selectively ripping up only the branch of a net that contains a conflict, along with any branches derived from it while leaving the unaffected parts intact. Although this strategy may, in some cases, result in suboptimal routing, potentially using more resources than strictly necessary it provides a significant improvement in execution speed, which as previously reported, is one of the main goals of Routino, following the specific of AMD Routing Contest [4].

I. Cost Function

During the routing part, Routino uses A* with the following formula:

$$C = c + h * 4$$

Where c is the accumulated cost of all the resources used to reach that tile, and h is the Manhattan distance between the tile and the sink tile. The heuristic cost is initially scaled by a factor of four to encourage a depth-first search behavior during the early iterations, progressively shifting the focus towards finding a valid solution in later stages. Different scaling factors were evaluated, and a multiplier of four was found to offer the best trade-off between routing speed and solution quality.

For resource cost estimation, Routino adopts the standard Pathfinder cost function, with a base resource cost of 1. The present congestion factor is initialized to 1 and doubled after each iteration, up to a maximum value of 256.

IV. Results

The effectiveness of the developed router has been validated on an XCVU3P Ultrascale+ device. We compare Routino with RWroute and Vivado using benchmarks of different sizes to determine the advantages in memory utilization, routing solution, performance, and elapsed time. Routino and RWroute rely on FPGAIF format as input files to provide the logic netlist with placement and the logic nets to

route. Differently, Vivado relies on a proprietary Design Checkpoint (.dcp) format. However, RapidWright provides a converter to port from .dcp to FPGAIF and back. We used Vivado in non-project mode from the command line to ensure that no additional memory was allocated for the GUI and that no caching was made before starting the routing.

A. Benchmark Circuits

We evaluated the performance of Routino on a set of benchmarks including designs from ITC99 [12], LogicNets [13], VTR [14], and Koios 2.0 [15] suites. The proposed benchmark suite includes designs with different dimensions, complexity, and application domains. Table I reports the list of circuits with a brief description. In Table II, we provide an overview of the benchmark in terms of dimensions and routable elements.

TABLE I. BENCHMARKS CIRCUITS

Design	Suite	Brief Description
b12	[12]	1-player game (guess a sequence)
b14	[12]	Subset of a Viper processor
b15	[12]	Subset of an 80386 processor
jscl	[13]	Jet Substructure Classification L
blob_merge	[14]	Image Processing
diffeq1	[14]	Mathematical Computations
diffeq2	[14]	Mathematical Computations
sha	[14]	Cryptographical Computations
stereovision0	[14]	Computer Vision
stereovision1	[14]	Computer Vision
stereovision2	[14]	Computer Vision
spmv	[15]	Sparse matrix-vector multiplication
test	[15]	Dummy design for regression testing
Bwave-like 1	[15]	Microsoft-Brainwave-like (large)
Bwave-like 2	[15]	Microsoft-Brainwave-like design (small)
lenet	[15]	Lenet Accelerator
eltwise_layer	[15]	Elementwise matrix operations

TABLE II. BENCHMARK CIRCUITS METRICS

Design	Primitives [#]	BRAMs [#]	DSPs [#]	Nets [#]
b12 [12]	364	0	0	366
b14 [12]	1,436	0	0	1,639
b15 [12]	2,156	0	0	2,645
jscl [13]	37,013	0	0	2,340
blob_merge [14]	7,412	0	125	9,903
diffeq1 [14]	1,154	0	9	1,753
diffeq2 [14]	626	0	9	1,105
sha [14]	2,434	0	0	2,655
stereovision0 [14]	13,534	0	0	3,403
stereovision1 [14]	29,697	0	0	4,793
stereovision2 [14]	22,992	0	246	3,242
spmv [15]	17,245	3	0	3,867
test [15]	747	0	4	342
bwave-like 1 [15]	33,180	292	1,472	15,303
bwave-like 2 [15]	47,085	170	328	12,284
lenet [15]	11,701	0	0	247
eltwise_layer [15]	21,910	36	0	4,524

B. Evaluation Metrics

We analyzed the performance of Routino comparing it with Vivado and RWroute solutions on different metrics, running on a workstation equipped with an Intel i9-9900K processor with a working frequency of 5 GHz and 32 Gb of DDR4 RAM.

Firstly, memory consumption is evaluated. We measured the memory consumption of routers by recording the peak memory usage. Peak memory usage has been measured using the ***usr/bin/time*** Linux program for Routino and RWroute, while Vivado provides this information autonomously at the end of the routing step.

The second metric is the quality of the routing, based on used resources (e.g. routed PIPs) and the delay of the critical path.

Finally, we measured the routing time. The metric has been extracted for Routino and RWroute using ***usr/bin/time*** Linux program, while Vivado provided the metrics autonomously. This metric is usually referred to as *Wall-clock*

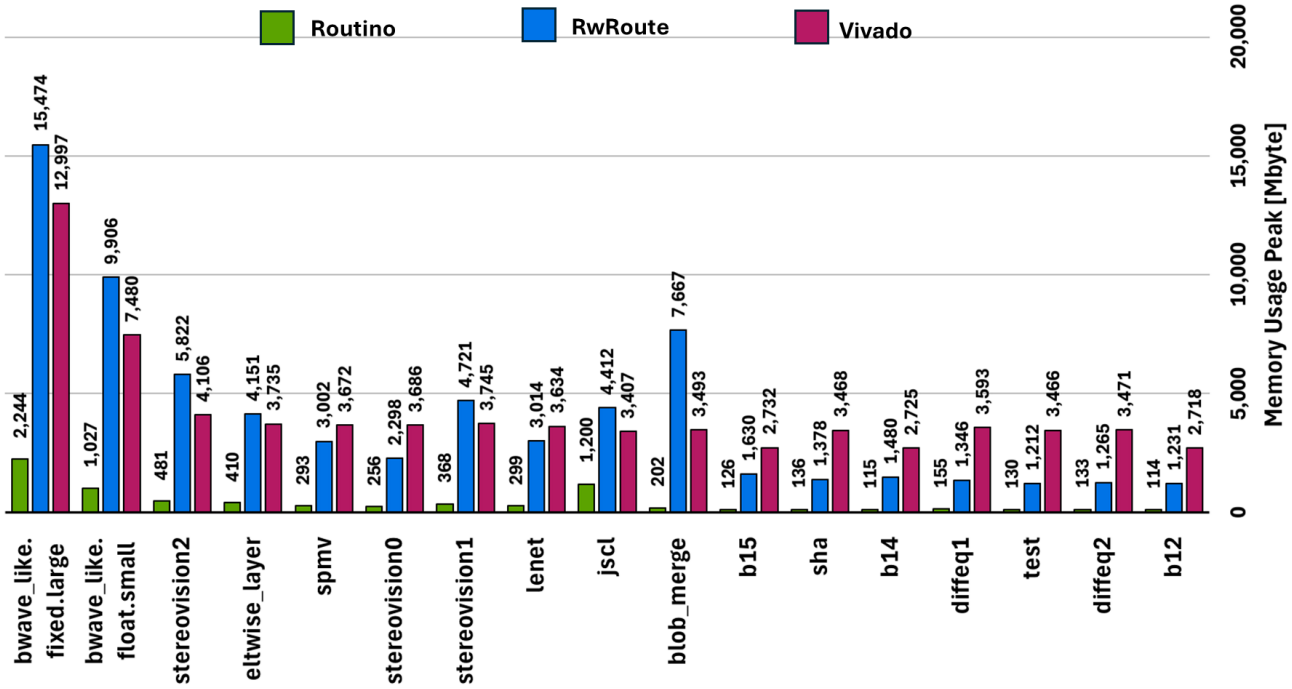

Fig. 5. Comparison of memory consumption metric among Routino (proposed), RWRoute, and Vivado. Smaller is better.

time or *time-to-solution,* and it is the time elapsed by the program to find a routing solution. In all the evaluated metrics, smaller is better.

C. Memory utilization

Regarding the memory utilization metric, Routino performed significantly better than both RWRoute and Vivado for all the benchmarks, with an average memory reduction against both of them of over 90%. Such a result is obtained by exploiting redundancy to describe the RRG. Fig. 5 shows a detailed comparison of all the benchmarks. A key aspect impacting also time-to-solution is that the RRG representation can be easily and completely kept in memory. Differently, the need for traditional routers to continuously deallocate and allocate memory for parts of RRGs or to load from memory large graph representation on demand greatly worsens the performance.

D. Routing Delay

In the comparison, Routino generally performed competitively against Vivado and RWRoute in terms of used PIPs and critical path delay, achieving better metrics on some designs. However, on average it increased the delay of about 20% against both Vivado and RWRoute and a similar increase in PIP usage. However, we want to emphasize that if we categorize critical path delay increase for design size, the average increase in delay is below 1% when considering only large designs, around 2% for medium designs, and about 24% for small designs. Such data show how Routino can produce competitive routing solutions for complex designs in a short time. The detailed comparison is reported in Fig. 6. To improve clarity when analyzing the PIP usage, we grouped the circuits into three categories represented with different scales, named small, medium, and large based on the number and average length of nets. The larger critical path delay is a direct consequence of Routino's design, which deliberately

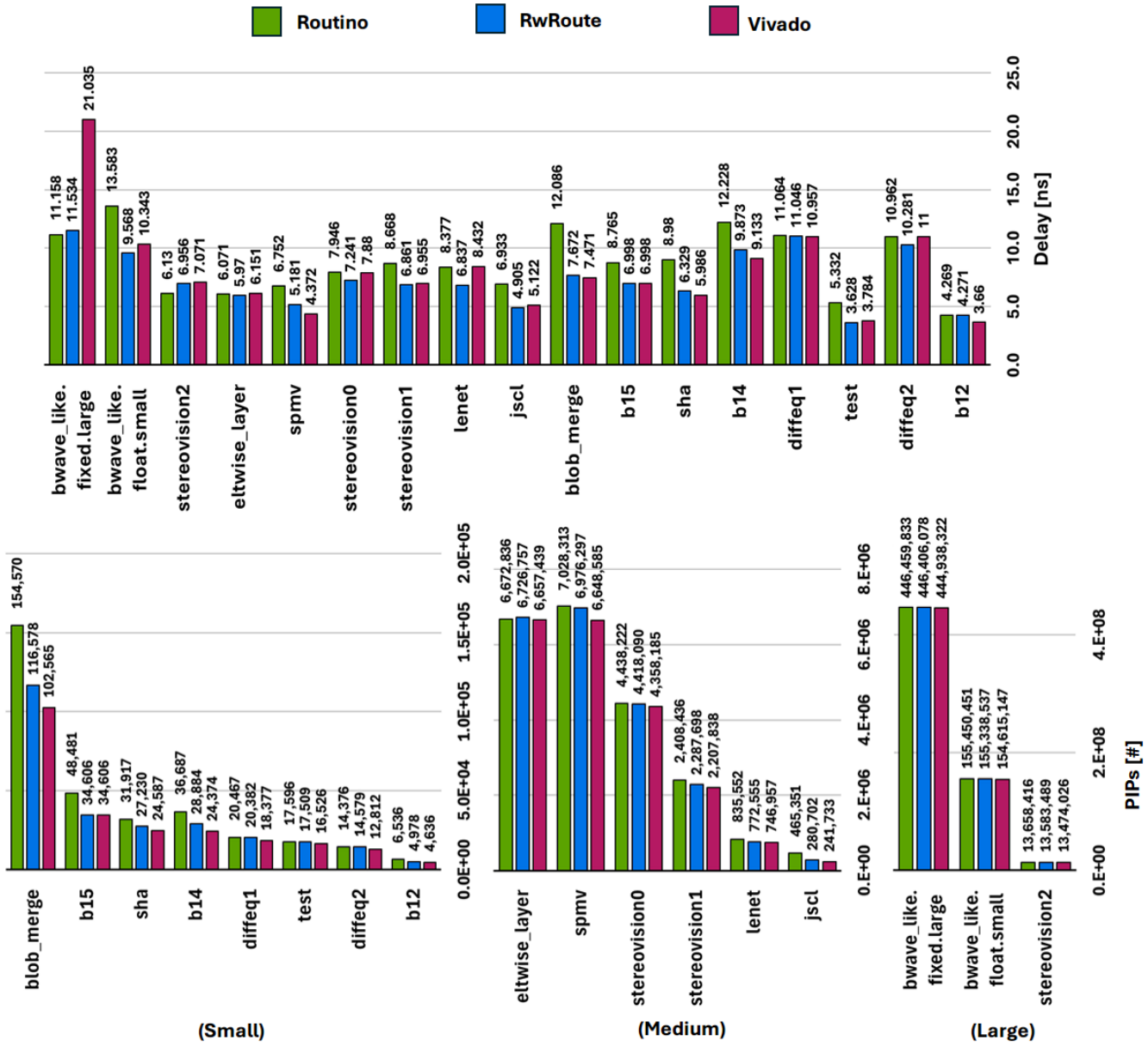

Fig. 6. Comparison of critical path delay (top) and used routing resources (bottom) among Routino (proposed), RWRoute, and Vivado. Smaller is better.

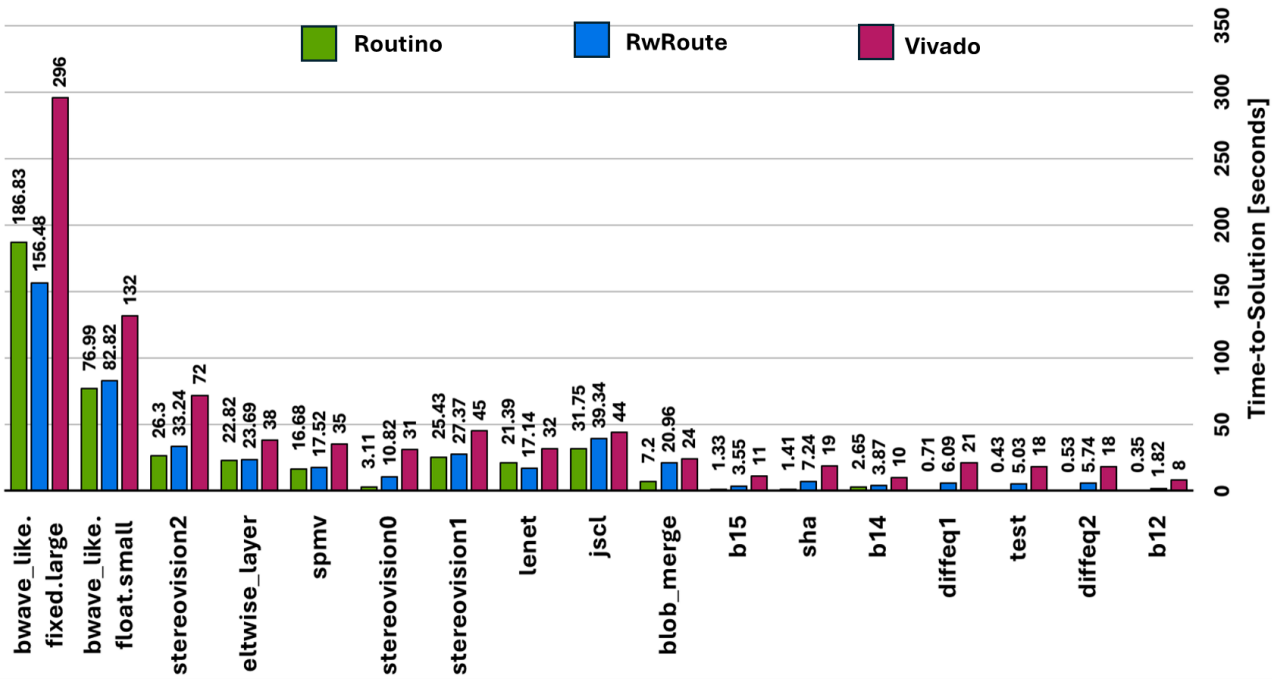

Fig. 7. Comparison of time-to-solution metric among Routino (proposed), RWRoute, and Vivado. Smaller is better.

prioritizes time-to-solution and memory efficiency over critical path optimization, following the objectives set by the 2024 AMD worldwide routing contest. Nonetheless, delay and resource usage remain important metrics. Routino adopts a greedy routing strategy, aiming to complete the routing quickly by avoiding costly re-routing steps and favoring unused resources whenever possible. While this approach significantly reduces routing time, it tends to overuse available resources and slightly worsens the delay, especially for small designs. A promising direction to mitigate this overhead would be to integrate alternative algorithms that better balance critical path optimization with routing speed.

E. Time-to-Solution

Regarding time-to-solution, Routino presents a significant speed-up compared to RWRoute and Vivado. Routino can route on average, about 40% faster than RWRoute and about 66% faster than Vivado. The detailed comparison is reported in Fig. 7. When considering only medium and large designs the speed-up is still significant on average, with a speed-up of about 20% and 50%, respectively. About this metric, we want to highlight that the current version of Routino works with a single thread, while Vivado and RWroute execute using multithreading. Routino may implement multithreading in the future, with all the performance benefits of it. Not least, the single-threaded version of Routino already outperforms in terms of time-to-solution.

V. Conclusion and Future Works

We presented Routino as an efficient and scalable solution for FPGA routing. It addresses the needs of new FPGA tools, prioritizing fast over optimal routing solutions, still achieving good routing quality results for large benchmark designs. We highlighted how it is possible to achieve great performance improvement mainly thanks to a novel representation of RRG exploiting FPGA intrinsic redundancy. In the future, we want to investigate the multithreading in Routino to improve speed-up and reduce the critical path delay.

References

[1] F. Mo, A. Tabbara and R. Brayton, A Force-Directed Maze Router, Department of EECS, University of California at Berkeley.

[2] R.Tessier, Negotiated A* Routing for FPGAs, in Proceedings of the Fifth Canadian Workshop on Field-Programmable Devices, 1998

[3] L. McMurchie and C. Ebeling, PathFinder: A negotiation-based Performance-Driven Router for FPGAs, ACM FPGA Symp. 1997.

[4] AMD Runtime-First FPGA Interchange Routing Contest, https://xilinx.github.io/fpga24_routing_contest/. Accessed: July 2024.

[5] C. Lavin and A. Kaviani, "RapidWright: Enabling Custom Crafted Implementations for FPGAs," 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 2018, pp. 133-140.

[6] Vaughn Betz and Jonathan Rose. Vpr: a new packing, placement and routing tool for fpga research. In Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL '97, 213–222. London, UK, 1997. Springer-Verlag.

[7] M. Shen, N. Xiao and G. Luo, "Dependency-Aware Parallel Routing for Large-Scale FPGAs," 2017 IEEE International Conference on Computer Design (ICCD), Boston, MA, USA, 2017, pp. 249-256, doi: 10.1109/ICCD.2017.45.

[8] D. Wang, Z. Duan, C. Tian, B. Huang and N. Zhang, "ParRA: A Shared Memory Parallel FPGA Router Using Hybrid Partitioning Approach," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 830-842, April 2020, doi: 10.1109/TCAD.2019.2901243.

[9] T. Martin, D. Maarouf, G. Grewal and S. Areibi, "A High-Performance Routing Engine for Large-Scale FPGAs," 2024 34th International Conference on Field-Programmable Logic and Applications (FPL), Torino, Italy, 2024, pp. 53-59, doi: 10.1109/FPL64840.2024.00017.

[10] Chin, S. Y. L. and Wilton, S. J. E. “Static and dynamic memory footprint reduction for FPGA routing algorithms”. ACM Trans. Reconfig. Techn. Syst. 1, 4, Article 18 (January 2009), 20 pages. DOI: 10.1145/1462586.1462587.

[11] FPGA interchange schema, https://github.com/chipsalliance/fpga-interchange-schema Accessed: July 2024.

[12] RT-Level ITC 99 Benchmarks and First ATPG Results", IEEE Design & Test of Computers, July-August 2000 (DOI: 10.1109/54.867894)

[13] Umuroglu, Y., Akhauri, Y., Fraser, N. J., & Blott, M. (2020, August). LogicNets: Co-designed neural networks and circuits for extreme-throughput applications. In 2020 30th International Conference on Field-Programmable Logic and Applications (FPL)

[14] J. Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. ACM Trans. Reconfigurable Technol. Syst. 7, 2, Article 6 (June 2014), 30 pages. https://doi.org/10.1145/2617593

[15] A. Arora et al., "Koios 2.0: Open-Source Deep Learning Benchmarks for FPGA Architecture and CAD Research," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 42, no. 11, pp. 3895-3909, Nov. 2023, doi: 10.1109/TCAD.2023.3272582.