electronics

MDPI

Article

NXRouting: A GPU-Enhanced CAD Tool for European Radiation-Hardened FPGAs

Andrea Portaluri [1,*], Sarah Azimi [1], Andrea Saracino [1], Luca Sterpone [1], Alp Kilic [2] and Damien Dupuis [2]

[1] Dipartimento di Automatica e Informatica (DAUIN), Politecnico di Torino, 10129 Turin, Italy; sarah.azimi@polito.it (S.A.); andrea.saracino@studenti.polito.it (A.S.); luca.sterpone@polito.it (L.S.)
[2] NanoXplore SAS, 92310 Sévres, France; akilic@nanoxplore.com (A.K.); ddupuis@nanoxplore.com (D.D.)
* Correspondence: andrea.portaluri@polito.it

Abstract: Field Programmable Gate Arrays (FPGAs) have witnessed an increase in space applications in the last years, mainly due to their cost-effective high-performances and flexibility. However, the susceptibility of these devices to radiation-induced effects when working in such an environment is well known. When common mitigation techniques are not sufficient to ensure the correct completion of a task, radiation-hardened FPGAs represent one of the most effective solutions. NanoXplore, in this context, is the first European developer of rad-hard FPGAs, which embed intrinsic high complexity in their architectures preventing the user from using or developing custom placement and routing algorithms. In this paper, we overcame these issues by proposing the first tool tailored to NanoXplore devices which allows the exploration of NanoXplore device architectures and routing of points through a Python interface. We developed a model that reflects the one used by the vendor, allowing the user to extract info about routes, nets and additional logic, otherwise unavailable. The tool also performs routing of points in the programmable logic, computing the optimal path. An implementation of the router on Graphic Processing Unit (GPU) is proposed to exploit the highly parallelizable nature of the problem. Finally, routing timing analyses on different benchmarks have been performed, improving the routing routine time.

Keywords: NanoXplore; radiation-hardened; FPGA; CAD; routing; GPU

Citation: Portaluri, A.; Azimi, S.; Saracino, A.; Sterpone, L.; Kilic, A.; Dupuis, D. NXRouting: A GPU-Enhanced CAD Tool for European Radiation-Hardened FPGAs. *Electronics* **2024**, *13*, 2803. https://doi.org/10.3390/electronics13142803

Academic Editor: Alexander Barkalov

Received: 18 June 2024
Revised: 12 July 2024
Accepted: 15 July 2024
Published: 16 July 2024

1. Introduction

The advent of the so-called New Space era brought a renewal of interest concerning deep space exploration. In this context, technologies have tried to keep pace to allow high computational power while limiting costs. Moreover, the increasing mission lifetimes require low-power consumption devices that can guarantee reliable computations. Among the several alternatives in the market, SRAM-based Field Programmable Gate Arrays (FPGAs) have captivated a good slice of the market thanks to their excellent performances over cost ratio. These devices exploit a matrix of reconfigurable logic resources that can be (re)programmed to execute virtually any digital circuit task at any moment. While FPGAs' performances are still not the same as Application Specific Integrated Circuits (ASICs), the versatility, low costs and time-to-market of these chips made several space industries choose them for on-board computers, communication and data acquisition in space [1–5].

However, deep space often represents a harsh environment, especially for Silicon-based technologies, where interactions with high-energy particles such as protons, neutrons and heavy ions can cause malfunctions in the circuitry. In particular, SRAM-based FPGAs are very susceptible to the corruption of the volatile memory where the data about the implemented design are stored and, thus, the correctness of its computational tasks [6]. Several techniques and approaches are available to mitigate these faults (e.g., Triple Modular Redundancy, Error Scrubbing, Module Isolation), allowing SRAM-based FPGAs to be safely adopted in these kinds of missions [6–8]. Anyway, when full coverage of these

errors is needed or in the case of extremely hard-working conditions, the said approaches may not be sufficient to guarantee flawless execution.

1.1. Motivations

Radiation-hardened FPGA chips are proposed as the main solution to this problem since they exploit the hardening of the architecture that lies behind the implemented design (e.g., intrinsic redundancy of resources, increased robustness of the memory cells, increased number of transistors in a logic cell) to ensure mitigation of radiation faults, in exchange for a slight increase in costs. Due to this, the majority of rad-hard FPGA vendors can claim the almost complete tolerance of their devices to radiation-induced effects [9]. In the context of rad-hard FPGA vendors and developers, NanoXplore has quickly gained popularity in the European market as the first European company to fully develop rad-hard chips, including the software toolchain. France-based fab-less company, NanoXplore counts several rad-hard FPGA technologies, both embedded and not, ranging from 65 nm up to 28 nm SRAM cells. Their NG-MEDIUM chip (65 nm CMOS) has also reached the ESCC QPL certification for space application issued by the European Space Agency (ESA), becoming the first space-qualified European FPGA [10].

Due to their rad-hard nature, NanoXplore FPGA architectures are complex and the routing of the available reconfigurable resources must obey several constraints. With the term routing, we refer to that phase of the FPGA design flow that usually follows the Synthesis and Placement, where the logic functions, associated with specific physical resources within the programmable logic, are connected in order to execute whatever Boolean function the circuit needs. While additional details and features on placement and routing are often available in commercial CAD tools, they are not yet supported for NanoXplore. For instance, at the time being, the possibility to force a fine-grained placement of the resources is not available as a feature within the Impulse tool as it would also require to drive far more complex interconnection routes manually (if available), often far from the designer knowledge or interest, that can fail the whole implementation if not realized correctly. For these reasons, the development of custom routing algorithms and third-party research for these architectures is tough and limited to the very little information available.

1.2. Main Contribution

To overcome these issues, in this paper, we propose NXRouting, the first in-house developed tool for the analysis of NanoXplore devices. It allows the user to fully explore the programmable logic and the routing of NanoXplore FPGAs using a simple Python interface. We developed a routing model in order to reflect the proprietary one in detail, showing internal routes and auxiliary logic, otherwise unavailable. These models were built using data extracted through NanoXplore's proprietary APIs, which we accessed through a collaboration with the vendor. Exploiting these data, the tool can route points in the programmable logic, computing and returning the optimal path if available and existing. A General-Purpose Graphic Processing Unit (GPGPU)-based implementation of the router is presented and timing analyses have been performed on several benchmarks.

The paper is structured as follows: Section 2 briefly overviews the related works, and Section 3 focuses on the model we developed to mimic the one of NanoXplore. Section 4 presents the tool, the configurations available, and the methodology we followed during its development. Section 5 presents the GPU implementation of the router and Section 6 shows the results of the experimental analysis we performed on the tool. Finally, Section 7 draws conclusions and discussion on the future extension of this work.

2. Related Works

The development of custom CAD tools has always been a flourishing field of study, aimed to add features or optimize tasks from the vendors' tools. In the case of Commercial-Off-The-Shelf (COTS) FPGAs (i.e., not radiation-hardened ones) and, in particular for the space application domain, several works have mainly focused on the computation of

routing paths or placement strategies for error mitigation. The authors in [11] propose a custom library in Python for the analysis of the bitstream, performing targeted fault injections on hardware for AMD Xilinx devices. Concerning the work in [12], the developed tool written in C language extracts data about the design reliability based on the place and route model of AMD Xilinx devices, proposing a hardened solution. The authors in [13] present a tool for the exploration of the routing architecture of AMD Xilinx Series 7 devices. Other works in literature are focused on bitstream manipulation, facilitating and automating partial reconfiguration of the device to scrub errors out [14,15]. Tools such as [16–19] have made attempts to reverse engineering bitstream, obtaining only partial results and targeting FPGA families that are three generations or more out of date compared to the current ones. However, the total absence of information about the NanoXplore proprietary bitstream format in the literature and official documentation made such an approach unexploitable. Moreover, these works are based on the Xilinx Design Language (XDL), which is no longer supported for newer devices, or complex interfaces with the AMD Xilinx CAD tool (i.e., Vivado Design Suite) making them, of course, limited to such architectures.

State-of-the-art approaches regarding FPGA routing include two phases: the modelization of the architecture as a routing graph, either directed or undirected (i.e., logic blocks are visualized as nodes while the available connections as edges), and the actual routing performed employing algorithmic choices. Several algorithms are found in the literature with the Maze, A* and Pathfinder being among the most prominent [20–22]. The Maze algorithm ensures the shortest path between two points in a grid. However, it does not consider how the path found might block subsequent nets. As a result, the algorithm's performance depends on the order of the nets. The A* algorithm improves efficiency by including the Manhattan distance to reduce the number of explored nodes. Finally, the Pathfinder balances path length and resource usage through an iterative process, where routing cost is dynamically adjusted based on congestion levels. As described later in the paper, the NanoXplore routing model differs from common ones and must obey several additional constraints, drastically reducing the number of usable algorithms and forcing us to implement less complex but still efficient ones.

A general trend in parallel routing algorithms is to enhance performance by implementing both fine-grained and coarse-grained parallelism techniques. Usually, fine-grained parallelism refers to parallel techniques for enhancing performance when building a single net, while coarse-grained techniques refer to techniques that embrace globally the routing process, trying to enhance performance when building multiple nets. Coarse-grained techniques usually employ spatial and architectural information to route different nets with the intent of minimizing the need for synchronizations. The authors in [23,24] use coarse-grained techniques that assign to each available CPU core a different set of nets, to be routed in parallel. This approach limits the parallelism to the number of CPU cores and parallelism might be exploited further with GPU-based approaches, where it is possible to massively parallelize the computation. The same limitation holds also to fine-grained techniques on CPU, as the one presented in [25].

Finally, the total absence of third-party tools for NanoXplore highlights the lack of data publicly available, limited to front-end info and features of the Impulse tool such as timing analysis and resource utilization reports. Due to these reasons, no other tool in the literature can perform placement and routing analysis of NanoXplore devices, making NXRouting the first to achieve such results.

3. Proposed Model

The following section will be dedicated to the description of the model we developed to overcome these limitations. The choices have been driven by details we extracted using NanoXplore APIs. Particular focus will be given to the routing of the core logic such as Look-Up Tables (LUTs) and Flip-Flops (FFs), since they represent the largest part of the routing effort in FPGAs.

3.1. Resources Hierarchy

The reconfigurable matrix is loosely based on the island-style FPGA model, where arrays of logic blocks are interposed with routing channels. Input and Output Buffers (IOBs), clock generators and, in the case of the newer devices (i.e., NG-ULTRA), interfaces for the processor are located at the edges of the programmable logic. The logic resources are spatially organized in a descendent tree manner, with the highest level being Plane followed by Zone, Network, Device, and Plug, as shown in Figure 1a. The Plane represents the whole programmable logic matrix and it is associated with the model of the FPGA chip (namely, Variant). The Zone is the first subset of resources, either logic (e.g., Tile and CGB) or for global routing purposes (e.g., Mesh), and interfaces (e.g., Fence, JTAG, IOB). The Networks are further subsets within the Zone while Devices and Plugs represent a physical resource and its pins. For instance, an input pin (e.g., Plug: I1) of a LUT (e.g., Device: LUT315) in the NG-MEDIUM variant can be described in Figure 1b. Finally, the Plugs of a Device can be mutually distinguished into Emitter (i.e., output pin) and Receiver (i.e., input pin). Moreover, it is useful to note that Zones names are unique (e.g., TILE [15 × 10] univocally identifies the Tile located in the 10th row and 15th column of the reconfigurable matrix), while Networks, Devices and Plugs names can be repeated among zones, although they will be still unique within the same one (e.g., with respect to the example shown in Figure 1b, a device named LUT315 appears in both zones TILE [15 × 10] and TILE [2 × 2] among the others but, within a given one, it is unique). In this way, a list as:

[zone_a, netwk_b, dev_c, plug_d]

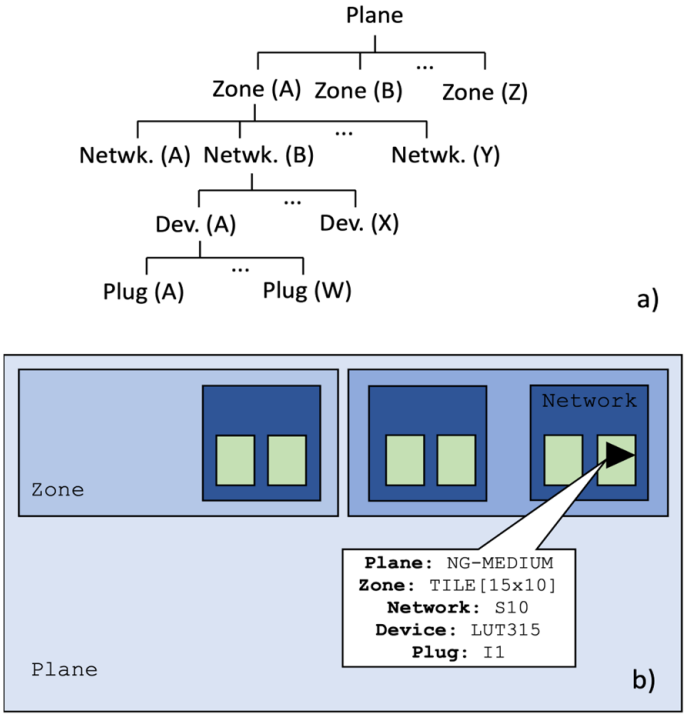

Figure 1. (**a**) Tree representation of the NanoXplore architecture; (**b**) Example of a point in the reconfigurable matrix identified through the hierarchy system.

Univocally identifies one and only one element. Figure 2 shows an example of Networks and Devices distribution inside of a Tile zone. This structure is repeated for each Tile in the programmable logic.

Figure 2. Graphical example of Networks (e.g., S10) and Devices (e.g., 289) inside a Zone (e.g., Tile).

3.2. Routing Model

Concerning the interconnection of the core logic, a clear distinction between routing inside and outside a Tile must be made. These represent the two cases of internal and general routing, cited in the NanoXplore documentation [26], and they will affect the timing differently. In the following subsection, we will be presenting how the model has been developed to cover these two scenarios.

When dealing with routing inside a Tile, we first need to identify and explain the Functional Element (FE), a hardwired coupling of a LUT and an FF, physically close to each other [26]. All common LUTs and FFs in the programmable logic are coupled in this way so that the output of the LUT must be connected with the input of the FF. Concerning the NG-MEDIUM architecture, Figure 3 presents a simplified view of the core logic inside a Tile and the FE architecture. The output of the FE (i.e., the output of the FF) can be then routed to further routing structures. This structural choice assumes then that the FF can be used as a proper memory element (i.e., D-type FF) or as a simple routing element (i.e., Bypass FF). In the same way, also LUTs can serve their scope as logic resources or behave as bypass elements, with an identity as a truth table. This scenario represents the vast majority of FE routing in the programmable logic, although the output of some LUTs can be also directly routed to CY blocks for high-speed chain computations or to other LUTs (only for NG-ULTRA and NG-ULTRA 300) to fulfill timing constraints. The last scenario has been neglected from the study for simplicity and will be further ignored.

Observations of timing analysis of intra-tile routing showed that the signal delay is almost independent of the coordinate of the FE within the Tile, meaning that the delay due to the routing of two consecutive blocks is the same as routing two opposite corners in the Tile. The latter excludes the possibility of having one of the most common routing scenarios where routing matrices are interposed between each logic block and directly connect different resources [26]; instead, this suggests a routing entity that is common for all the resources in the Tile. In order to ensure the routing reaches all of the four inputs of the LUTs and does not create conflicts by excluding any of these pins as the target of a route, four routing matrices have been added to each Tile, each of which is associated with one input of the LUTs. Within each one of these, the routing matrix has been modeled by two Devices, allowing the routing to be correctly shuffled in any desired way and to reach each FE of the Tile without creating conflicting routes. Figure 4 presents the adopted model (Figure 4b) while comparing it to a common routing model (i.e., the one adopted by most AMD Xilinx architectures, Figure 4a).

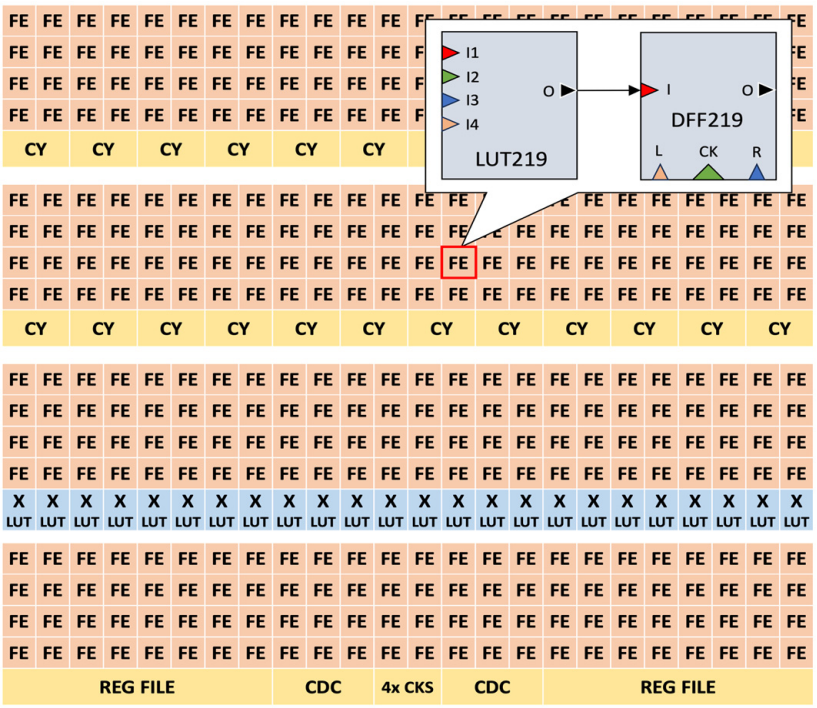

Figure 3. Detail of a Tile in the NG-MEDIUM architecture with focus on the FE and its components. The Tile includes also additional logic such as Carry logic (CY), High-performance LUT (X-LUT), Register File and others.

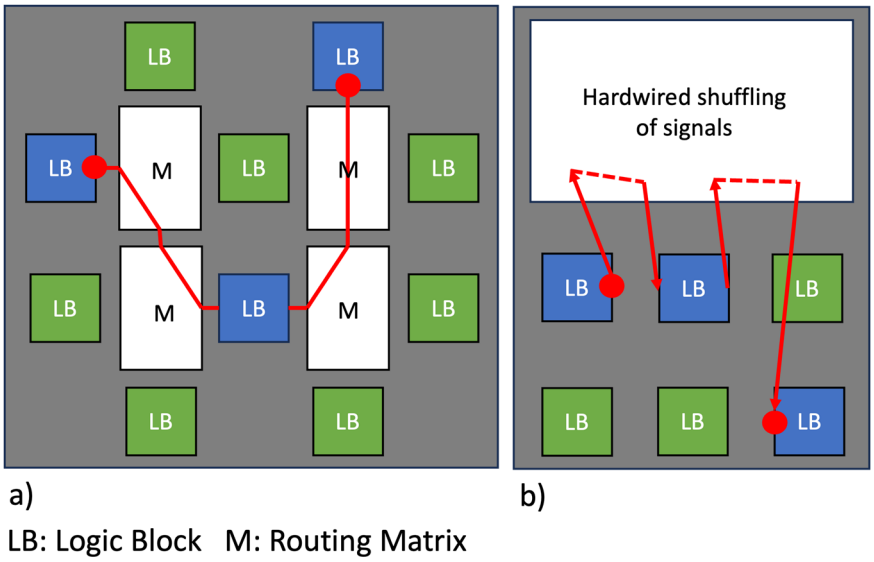

Figure 4. The figure explains the difference between the two routing models inside a Tile. (**a**) Routing by means of switch matrices, where the delay is dependent on the physical distances among the routed logic blocks; (**b**) Routing by means of shuffling matrix, where the delay only roughly depends on how many times the route enters the matrix.

As designs become more complex and require more logic to be routed, a single Tile may not contain enough resources to fulfill the scope. Therefore, NanoXplore refers to general routing when dealing with Tile-to-Tile signals [26]. In order to manage the access to these external routing channels from the Tile, two blocks have been modeled and added to it. Moreover, observing the displacement of Tiles and the zones that manage the general routing (namely, Meshes) in the configuration memory of NanoXplore devices, it is safe to assume that a Tile can be reached both from a Mesh located at the bottom and the one

located at the top. This assumption is also validated by the Impulse tool Graphic User Interface (GUI), where the Tile shows two sets of access points as well as two exiting points, both coupled in the upper and lower side of the Tile, as shown in Figure 5. When in the Mesh, the signal can now be routed globally crossing other Meshes with a span of several ones in the horizontal direction (depending on the Variant) and ± 1 in the vertical one. This solution allows the signal to reach any Mesh in the configurable plane starting from any point.

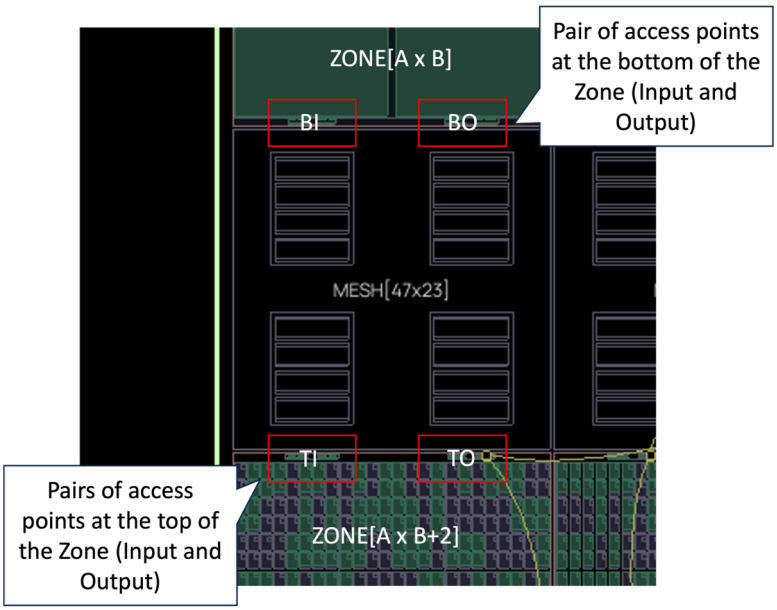

Figure 5. Floorplanning view from the Impulse tool. Access points at the top (TI and TO) and bottom (BI and BO) of two Zones allow the routes to enter the general routing channels through the Mesh.

4. NXRouting Tool

In this section, the NXRouting tool will be presented. In particular, the following paragraphs will explain how the model discussed before has been implemented and exploited to allow the tool to emulate the NanoXplore routing architecture. The final subsection also discusses the implemented routing feature among points in the configuration memory.

4.1. Database Management

According to the routing directives described before, a database of resources and available connections has been built for each variant. Each row of the database gives the coordinates of a source point and a target point in the naming convention described in Section 3.1. Therefore, the database includes eight columns (i.e., two pairs of four coordinates). These data have been extracted through C++ NanoXplore APIs and their size is dependent on the variant and the amount of resources within (e.g., ranging from 76.8 MB for NG-MEDIUM up to 1.32 GB for NG-ULTRA). To efficiently deal with the database exploration, two different loading configurations have been realized for the user to choose from. The first one, named Full Loading, performs the load of the entire contents of the database into cache memory, allowing the best performances in terms of computational speed. The drawbacks of this solution are an initial time overhead to load the full database (in the order of tens of seconds for the NG-MEDIUM up to tens of minutes for NG-ULTRA) and the limited size of cache memory available (some systems may have difficulties loading GBs of data). The second solution we have adopted is to load data only when requested (Request Loading); the system calls a load method that retrieves specific data from the database at runtime. This approach does not require any time overhead at start-up but it may involve several performance decreases during the routing exploration. Both solutions

are available, and the user must decide which one to use at launch time depending on the needs.

4.2. Data Organization and Structures

Once the database loading routine has been chosen, the system starts to fetch data accordingly. A top structure is created and associated with the Plane, given the variant as input argument. The Variant class couples the Plane with the respective database unless the latter is manually overwritten by the user. The plane contains a Python dictionary that has the first level of hierarchy as keys (i.e., Zones) and the objects themselves as values, as well as other useful getter methods (e.g., getName(), getID(), getZone(name), and so on). This approach is necessary in order to avoid any duplication of a class object referred to the same element, so it will be unique as well as its attributes. This structure is repeated for each level of the hierarchy, allowing an agile exploration of the whole programmable logic. Each object also contains the reference to its parent, so that it is possible to retrieve it at any moment. This interface fully integrates with the Python object-oriented programming paradigm, for instance, exploiting iterators that allow the user to use cycles and indexing on the returned objects. Figure 6 gives an example of the flexibility of the exploration from the Python command line using NXRouting.

```
>>> plane = Plane(variant_name = 'NG-MEDIUM')
>>> plane.getDatabase().getName()
'NG_MEDIUM.db'
>>> zone_0 = plane.getZone(zone_name = 'TILE[15x10]')
>>> netw_0 = zone_0.getNetwork(netw_name = 'S10')
>>> for device in netw_0.getDevices():
>>>         device.getName()
>>>
LUT315
LUT316
[...]
>>> netw_1 = zone_0.getNetwork('S10')
>>> netw_0.getZone() == netw_1.getZone()
True
```

Figure 6. Example of architectural exploration through pseudo-Python of the NXRouting tool. This feature mainly focuses on the user experience, presenting a well-known objected-oriented interface.

4.3. Routing Points

Among the architectural exploration, as stated before, a routing feature has been added to the tool in order to retrieve all the connection branches between two points in the programmable logic. When integrating the detailed model explained in Section 3.2 to this possibility, the tool allows the user to analyze a specific net; for instance, the observation of a route may suggest that a high delay is due to too many global jumps among meshes and so it might be solved by constraining the placement within closer tiles.

The problem of routing this specific architecture can be seen as a directed graph where each node has one or more directed edges (i.e., the direction of the edge is defined as apriori, and it does not allow the signal to travel arbitrarily) linked to other nodes. In our solution, Plugs represent the nodes and the connections listed in the databases identify the edges and their direction. However, each Plug cannot allow incoming and outgoing

branches since it is classified either as an emitter or receiver. To overcome this issue, the data structures defined in Section 4.2 may come in handy, returning the emitters associated with a Device and its receivers. This approach extends the concept of the node to an object that includes a Device, Emitters, and Receivers, although still allowing the user to route specific Plugs. For the routing algorithm, the Breadth-First Search (BFS) has been chosen due to its well-known structure and performance [27]. In particular, the BFS algorithm allows to search for a target by exploring all the nodes at the present depth of the graph before moving to the next depth level, in contrast to the Depth-First Search (DFS) that analyses a single node branch in its full depth before expanding to the next one. The implementation of the algorithm is described below in Algorithm 1. The implementation loosely follows the standard BFS, although it contains substantial differences. Each emitter will be searched through by popping it out from the queue list. A for loop iterates over every possible receiver that can be reached from the analyzed node and each emitter of the same device of that receiver will be added to the queue as well. The algorithm stops searching when the target receiver has been found. However, the standard algorithm faces some limitations. In particular, plain BFS is not able to manage graphs with loops as they may arise infinite cycles of search for such branches. To overcome this problem, three flags have been added to the Plugs: visited, queued, and routed. The visited one keeps track of which plug has already been completely searched while the queued flag identifies a plug already in the queue waiting to be searched. The routed flag, instead, has been added to allow multiple consequent routings and to know which node has already been taken by a net or is still vacant. At the end of the routing routine, the first two flags will be reset. The results are saved in a Net object that contains each branch the algorithm followed to reach the target (i.e., parent nodes in Algorithm 1). In Figure 7, a scheme presents the whole flow of the NXRouting described so far.

Algorithm 1. Algorithm for BFS of NanoXplore Architectural Graph

```
Input: source_emitter, target_receiver
queue = [source_emitter]
while len(queue) > 0 do
    node = queue.pop(0)
    if not node is routed then
        if not node is visited then
            for receiver in node.receivers do
                if not receiver = target_receiver then
                    if not receiver is visited then
                        for emitter in receiver.getDevice().getEmitters() do
                            if not emitter is queued then
                                update emitter parent nodes
                        queue.append(emitter)
                        emitter is queued
                else:
                    update node parent nodes
                    break from while
    node is visited
```

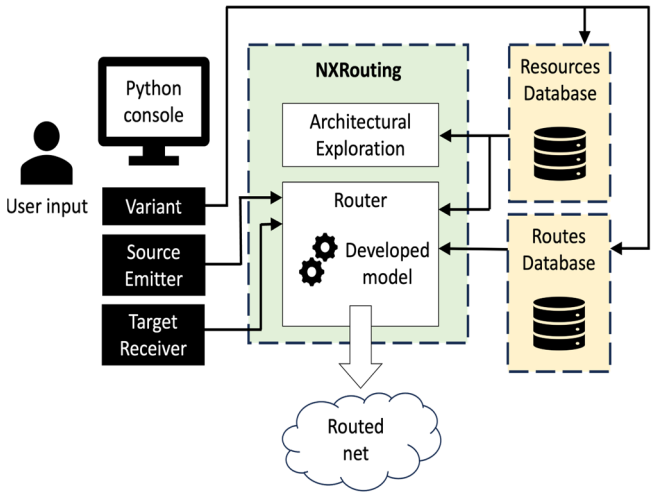

Figure 7. Scheme of NXRouting showing the features and the databases associated.

5. GPU Implementation

This section describes the implementation of the routing algorithm and the data structures within the Cuda-C environment by NVIDIA. In particular, two levels of parallelization have been exploited, namely fine and coarse-grained. The first one deals with the concurrent search among the nodes in the queue regarding Algorithm 1 while routing a single net. The coarse-grained, instead, manages the routing of more nets at once. The following subsections will present the two in depth.

5.1. Fine-Grained Parallelization

Some new data structures have to be introduced to support parallelization on GPU. These must be allocated in memory N times, where N is the number of nodes to explore in parallel, differently from the ones introduced in Section 4.2. The data structures are:

- *Local labeling array*: it marks nodes when they are discovered for the first time and keeps track of which nodes were already found in previous iterations.
- *Previous and next buffers*: a double buffer that at each iteration contains the indexes of new nodes discovered at the previous iteration, and the indexes of nodes discovered in the current iteration. At each new iteration, their role reverses.
- *Backtracking array*: when in the GPU kernel new reachable nodes are discovered, the history of previous nodes has to be saved. The *backtracking* array saves, for each discovered node, which node discovered it. When the sink is eventually found, the shortest path from source to target can be reconstructed, thanks to the information stored here (i.e., parent nodes in Algorithm 1).

Before starting the next iteration, the BFS swaps the pointers of the two buffers, next and previous, and clears the content of the next buffer preparing it for the next iteration. The clear is implemented without coping or removing data, but with an index telling the length of the valid content is zero. Avoiding any copy between the buffers and avoiding any deletion allows the algorithm to reach better performances according to the ping-pong buffering technique. When starting the second iteration, thanks to pointer swapping, the previous buffer will contain the set of nodes just discovered for the first time in the previous iteration, and the GPU kernel will launch as many threads as the number of new nodes previously discovered.

If a valid path is found, the iterations stop and the shortest path is reconstructed by checking the backtracking array traversing it backward and target to source. Local data structures can be either reinitialized for other nets or deallocated.

5.2. Coarse-Grained Parallelization

The last section explained the steps to build a GPU-based BFS algorithm to find a route from source to sink. It is possible to maximize the percentage of GPU usage even further by modifying the previous algorithm to support the computation of BFS of multiple nets concurrently.

When routing multiple nets in parallel, one of the main issues is that when one of the nets finds its shortest path, it needs to remove those resources from the available pool. After marking those resources as no longer available, the other nets that are computing their own route at the same time need to prune all the new nodes that were discovered after traversing one of the nodes just removed from the pool, because they would walk a path no longer available. This check would cause a significant performance loss, that could scale even worse by increasing the number of routes in parallel. The worst case would be represented by spending more time checking if nodes were invalidated than searching the routes themselves. To avoid this computation, it is possible to leverage information about the FPGA architecture, described in Section 3.1. The coarse-grained technique consists of dividing the list of nets into N+1 different sets, with N being the number of Tiles. One set contains nets that have source and sink in different Zones. The other N sets contain nets with source and target within the same Tile. Each set will route its nets sequentially, resulting in N+1 sequential flows of routing. However, N of these sets route nets that use different routing resources, allowing us to route them in parallel without checking if the resources taken can alter the other BFSs. Thus, when one of the N-1 sets removes from the pool some available resources, the resources removed will only be resources that might have been used inside that set. The N_{i+1} set contains every other net, (i.e., nets with source and sink not belonging to the same Tile) or with one or both of them being inside a Zone that is not a Tile. This set cannot be routed concurrently with the others because it may remove available resources that are in common with the other sets. Because of this, it can either be the first to be computed, before the other N sets, or the last one, after the other N sets finished their routing.

6. Experimental Results

This section presents the analyses and the results we collected in order to quantify the performances of our tool. First, profiling the two database loading configurations has provided an efficient way to discriminate the choice according to the feature the user wants to utilize. The time the system has taken to fetch each level of hierarchy has been presented in Table 1 for two variants (NG-MEDIUM and NG-ULTRA, respectively).

Table 1. Response Time of Start-Up and Getter Methods for the Loading Configurations.

Config.	Start-Up [µs]		getZone [µs]		getNetwork [µs]		getDevice [µs]		getEmit. [µs]	
	NG-MED	NG-ULT	NG-MED	NG-ULT	NG-MED	NG-ULT	NG-MED	NG-ULT	NG-MED	NG-ULT
Full Load.	19×10^6	452×10^6	4.768	5.323	4.034	4.873	4.733	5.543	4.768	5.323
Req. Load.	-	-	189.452	378.574	245.829	201.551	387.002	568.719	639.402	739.321

As stated in Section 4.1, the Full Loading configuration mainly affects the start-up timing as all the data are loaded in memory. In particular, the NG-ULTRA analysis takes up to 8 min to complete the loading. However, this overhead is required only at the start of the tool and it does not affect any subsequent computation. The Request Loading configuration does not need any start-up overhead although getters take significantly more time compared to the Full Loading. A slight increase in the time as the hierarchy level goes from Zone to Plug has also been registered in this configuration. This is due to the more complex queries and filters that the system demands from the database as the hierarchy level of the objects decreases.

Moreover, an analysis of the timing required for NXRouting to fully route several benchmarks using the routePoints method on NG-MEDIUM architecture has been performed. Four benchmarks have been selected from the ITC'99 benchmark suite [28], in

particular B03, B06, B09, and B12. These designs have been chosen for their well-known architecture, and behavior and for covering a wide section of the routing complexity spectrum. Table 2 shows the number of nets to be routed for each benchmark.

Table 2. Number of Nets to be routed for each Benchmark.

Benchmark	Number of Nets [#]
B03	253
B06	68
B09	284
B12	1688

Results about the routing time have been collected from three different routing configurations exploiting parallelization at various levels. The first configuration, *sequential*, implements no parallel computation performing routing of signals in series. The second and third configurations exploit fine and coarse-grained parallelization described in the previous sections. Unfortunately, no other third-party router is available in the literature for comparison at the time being. Table 3 presents the obtained results. The tool has been implemented on a GPU NVIDIA Jetson Nano.

Table 3. Performance Comparison among Routers Configurations on NG-MEDIUM.

Benchmark	Routing Time [s]			
	B03	**B06**	**B09**	**B12**
Sequential Router	143.17	62.73	300.05	433.14
Fine-Grained Router	0.81	0.47	0.89	4.62
Fine-Coarse-Grained Router	0.60	0.47	0.58	2.64

The results show a decrease in routing time using the Fine-Coarse-Grained configuration with respect to the other ones. However, the timing still seems to scale on the number of nets to route. This is mainly due to the unfeasibility of issuing a number of kernels equal to the amount of nets (i.e., achieving an embarrassingly parallelization of the problem).

7. Conclusions

Among the rad-hard FPGA vendors, NanoXplore is the first European developer of SRAM-based radiation-tolerant devices. Its market is continuously growing and the users started to feel the necessity to develop their own placement and routing algorithms to fulfill custom needs. However, NanoXplore models are very different from the most common ones due to several architectural constraints that the devices present. For this reason, common placement and routing approaches cannot be applied. Moreover, the lack of info about the routing structure negatively affects any third-party research effort.

In this paper, we presented NXRouting, the first Python-based tool that allows the user to easily explore the NanoXplore architecture, its hidden modules and resources exploiting data coming from the C++ vendor APIs. Therefore, the designer will be able to analyze and extract the architectural details in order to validate custom placement algorithms. Two different start-up configurations are available to choose from in order to focus on performances or timing, and a profiling analysis has been performed in the paper. Moreover, the tool presents a routing feature to drive nets from and to any points in the reconfigurable plane according to the Breadth-First Search algorithm. This allows the extraction of the branches and resources involved and to validate the routability and feasibility of custom algorithms and implementations, emulating the routing model from NanoXplore in detail. A GPU implementation of the tool and the algorithm is proposed in the paper and routing timings have been extracted for several benchmarks.

Author Contributions: Conceptualization, A.P.; methodology, A.P.; software, A.P. and A.S.; validation, A.P.; writing—original draft preparation, A.P.; writing—review and editing, S.A.; supervision, S.A, L.S., A.K. and D.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Restrictions apply to the availability of these data. Data were obtained from NanoXplore SAS and are available from the authors with the permission of NanoXplore.

Acknowledgments: We would like to acknowledge NanoXplore for giving us the possibility to extract data from their proprietary libraries and APIs.

Conflicts of Interest: Authors A.K and D.D. were employed by the company NanoXplore. The remaining authors declare that the research was conducted in absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. León, A.F. *Trends and Patterns of ASIC and FPGA Use in European Space Missions*; ESA-ESTEC: Noordwijk, The Netherlands, 2013.
2. Kok, C.L.; Siek, L. Designing a Twin Frequency Control DC-DC Buck Converter Using Accurate Load Current Sensing Technique. *Electronics* **2024**, *13*, 45. [CrossRef]
3. Teo, B.C.T.; Lim, W.C.; Venkadasamy, N.; Lim, X.Y.; Kok, C.L.; Siek, L. A CMOS Rectifier with a Wide Dynamic Range Using Switchable Self-Bias Polarity for a Radio Frequency Harvester. *Electronics* **2024**, *13*, 1953. [CrossRef]
4. Kok, C.L.; Tang, H.; Teo, T.H.; Koh, Y.Y. A DC-DC Converter with Switched-Capacitor Delay Deadtime Controller and Enhanced Unbalanced-Input Pair Zero-Current Detector to Boost Power Efficiency. *Electronics* **2024**, *13*, 1237. [CrossRef]
5. Kong, J.; Siek, L.; Kok, C.-L. A 9-bit body-biased vernier ring time-to-digital converter in 65 nm CMOS technology. In Proceedings of the IEEE International Symposium on Circuits and Systems(ISCAS), Lisbon, Portugal, 24–27 May 2015; pp. 1650–1653. [CrossRef]
6. Wirthlin, M. High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond. *Proc. IEEE* **2015**, *103*, 379–389. [CrossRef]
7. Portaluri, A.; De Sio, C.; Azimi, S.; Sterpone, L. A New Domains-based Isolation Design Flow for Reconfigurable SoCs. In Proceedings of the 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), Turin, Italy, 28–30 June 2021.
8. Deepa, M. An Improvised Voter Architecture For TMR With Reduced Area Overhead. In Proceedings of the Third International Conference on Intelligent Computing Instrumentation and Control Technologies (ICICICT), Kannur, India, 11–12 August 2022; pp. 1001–1007.
9. NanoXplore. From Radiation Hardening to BRAVE FPGA devices. In *RADSAGA Initial Training Event*; NanoXplore: Geneva, Switzerland, 2017.
10. European Space Components Coordination. *ESCC Qualified Part List (QPL) ESCC/RP/QPL005-246 (REP 005)*; ESA: Paris, France, 2024.
11. De Sio, C.; Azimi, S.; Sterpone, L.; Merodio Codinachs, D.; Decuzzi, F. PyXEL: Exploring Bitstream Analysis to Assess and Enhance the Robustness of Designs on FPGAs. In Proceedings of the 2023 19th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), Funchal, Portugal, 3–5 July 2023; pp. 1–4.
12. Azimi, S.; Du, B.; Sterpone, L.; Merodio Codinachs, D.; Cattaneo, L. SETA: A CAD Tool for Single Event Transient Analysis and Mitigation on Flash-Based FPGAs. In Proceedings of the 2018 15th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), Czech Republic, Prague, 2–5 July 2018.
13. Petersen, M.B.; Nikolić, S.; Stojilović, M. NetCracker: A Peek into the Routing Architecture of Xilinx 7-Series FPGAs. In Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Array, Virtual Event, 28 February–2 March 2021.
14. Guccione, S.A.; Levi, D.; Sundararajan, P. JBits: A Java-based interface for reconfigurable computing. In Proceedings of the Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD), Laurel, MD, USA, 28–30 September 1999.
15. Pham, K.D.; Horta, E.; Koch, D. BITMAN: A tool and API for FPGA bitstream manipulations. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017.
16. Haroldsen, T.; Nelson, B.; Hutchings, B. RapidSmith 2: A Framework for BEL-level CAD Exploration on Xilinx FPGAs. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15), New York, NY, USA, 22–24 February 2015; pp. 66–69.
17. Lavin, C.; Kaviani, A. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In Proceedings of the IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 29 April–1 May 2018; pp. 133–140.
18. Zhang, T.; Wang, J.; Guo, S.; Chen, Z. A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code. *IEEE Access* **2019**, *7*, 38379–38389. [CrossRef]

19. Benz, F.; Seffrin, A.; Huss, S.A. Bil: A tool-chain for bitstream reverse-engineering. In Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, 29–31 August 2012; pp. 735–738.
20. Mo, F.; Tabbara, A.; Brayton, R.K. A force-directed maze router. In Proceedings of the IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281), San Jose, CA, USA, 4–8 November 2001; pp. 404–407.
21. Tessier, R. Negotiated A* Routing for FPGAs. In Proceedings of the 5th Canadian Workshop on Field Programmable Devices, Montréal, QC, Canada, 7–10 June 1998.
22. McMurchie, L.; Ebeling, C. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays, Napa Valley, CA, USA, 12–14 February 1995; pp. 111–117.
23. Shen, M.; Luo, G. Accelerate FPGA routing with parallel recursive partitioning. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 2–6 November 2015.
24. Chan, P.; Schlag, M. Acceleration of an FPGA router. In Proceedings of the Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186), Napa Valley, CA, USA, 16–18 April 1997.
25. Gort, M.; Anderson, J. Deterministic multi-core parallel routing for FPGAs. In Proceedings of the International Conference on Field-Programmable Technology, Beijing, China, 8–10 December 2010.
26. NanoXplore. NanoXplore Wiki. Available online: https://nanoxplore-wiki.atlassian.net/wiki/spaces/NAN/overview?mode=global (accessed on 2 June 2024).
27. Palanisamy, V.; Vijayanathan, S. Cluster Based Multi Agent System for Breadth First Search. In Proceedings of the 20th International Conference on Advances in ICT for Emerging Regions (ICTer 2020), Colombo, Sri Lanka, 4–7 November 2020.
28. Davidson, S. ITC'99 Benchmark Circuits—Preliminary Results. In Proceedings of the International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034), Atlantic City, NJ, USA, 30 September 1999; p. 1125.