FPGA-Assisted Deterministic Routing for FPGAs

Dario Korolija
Department of Computer Science
ETH Zurich
Zurich, Switzerland
dario.korolija@inf.ethz.ch

Mirjana Stojilović
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
mirjana.stojilovic@epfl.ch

***Abstract*—FPGA routing is one of the most time-consuming steps of FPGA compilation, often preventing fast edit-compile-test cycles in prototyping and development. There have been attempts to accelerate FPGA routing using algorithmic improvements, multi-core or multi-CPU platforms. Instead, we propose porting FPGA routing to a CPU+FPGA platform. Motivated by the approaches used in FPGA-accelerated graph processing, we propose and implement three acceleration strategies: (1) reducing the number of expensive random memory accesses, (2) parallel and pipelined computation, and (3) efficient hardware priority queues. To test and evaluate the router performance, we implement it on DE1-SoC, a mid-end ARM+FPGA platform of Intel. Our router works and produces good quality results. Moreover, we succeed in accelerating the software router running on the embedded ARM. However, when compared to the latest VPR router running on a powerful Intel Core-i5 CPU, our CPU+FPGA router is slower. This is not unexpected, given the limited performance of the chosen hardware platform. Since this design can easily be ported to newer and higher-end CPU+FPGA systems, we estimate the performance it could achieve; the results indicate that a non-negligible speedup over the software-only router could indeed be obtained.**

***Index Terms*—FPGA, routing, parallelism, hardware acceleration, PathFinder**

I. Introduction

With big cloud-service providers introducing FPGAs into their portfolio, Microsoft deploying FPGAs in all their servers, and Intel releasing their Atom CPU series E6x5C with FPGA fabric in the same package, FPGAs are in the spotlight these days. Yet, not only they remain difficult to program, but also compiling an industrial-size design may take prohibitively long time (hours, days).

FPGA routing is one of the most time-consuming steps of FPGA compilation. The commercial and academic FPGA compilation tools, such as Verilog-to-Routing (VTR) [1], typically use the PathFinder [2] algorithm for FPGA routing. This work does not attempt to accelerate PathFinder in software, but to embrace the emerging trend of bringing FPGAs and CPUs closely together by designing an FPGA router suitable for a CPU+FPGA acceleration platform.

The paper is organized as follows. After presenting related work (Section II), we define FPGA routing and describe PathFinder (Sections III). In Sections IV and V, we present our acceleration strategies and our maze routing algorithm, respectively. The details of the hardware accelerator implementation are given in Section VI, which is followed by experimental evaluation and discussion (Section VII).

II. Related work

FPGA-based accelerators are known to help speed-up graph processing [3]–[6]. The algorithms they target are single-source shortest path (SSSP), weakly connected component (WCC), minimum spanning tree (MST), etc. These require the entire graphs to be read and processed. However, traversing the entire graph is unneeded for FPGA routing and it would considerably affect the router performance. Our design approach is inspired by the work of Dai et al. [6]. They propose to segment the graph vertices into small intervals, which can fit the FPGA on-chip memory, while streaming edges from the off-board storage. This, they claim, makes their solution capable to scale to graphs with billions of vertices and edges. We build on their idea but adapt it to the FPGA routing problem.

Accelerating FPGA routing using FPGAs was proposed by DeHon et al. too [7]. However, their approach is fundamentally different as they modify the underlying FPGA fabric to assist in routing.

III. FPGA Routing

The primary data structure representing FPGA routing resources is the directed *Routing Resource Graph (RRG)* $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each vertex $v \in V$ represents wires and pins that are internal to an FPGA. Each edge $e_{ij} \in E$ represents a programmable connection point between a pin and a wire segment, or a programmable routing switch between two wire segments. In modern FPGA architectures, switches are buffered and thus unidirectional. A signal i to route through G forms a net $N_i = (s_i, \{t_{i,1}, t_{i,2}, ..., t_{i,m}\})$, where s_i is the source vertex and $\{t_{i,1}, t_{i,2}, ..., t_{i,m}\}$ are the sinks. The solution to the routing problem of the net N_i is a set of paths from the source s_i to all the net sinks; these paths form a directed *routing tree* $RT(N_i) \subset G$. Routing is successful if all final routing trees are disjoint in G.

A. PathFinder

The most common academic and commercial FPGA routers use PathFinder algorithm [2]. PathFinder implementation is a triple-nested loop. The outer loop (*all-net router*), invokes the middle loop (*signal router*), for all signals i to be routed. If there is no congestion, i.e., no routing trees share routing resources, or if a user-defined number of iterations is exceeded, PathFinder terminates. Otherwise, the second-order (also called historical or accumulated congestion) costs of all congested nodes are updated.

Each signal router iteration starts by riping-up the existing routing tree $RT(N_i)$ of a net N_i. As a consequence, the occupancy of nodes in $RT(N_i)$ is decremented and their first-order (also called present congestion) costs updated accordingly. Then, it invokes the maze expansion to route the net N_i. Once new routing tree is created, the present congestion costs of all its nodes are updated.

Maze expansion traverses the RRG, starting from the source node of a net. It initializes the routing tree $RT(N_i)$ with the source node. Then, it expands the source node, i.e., uncovers all its neighbors and stores them in a priority queue (PQ) sorted by their costs. In each subsequent maze expansion iteration, the lowest-cost vertex v_{min} is extracted from the PQ. If v_{min} is a sink of the net N_i, a path is constructed by invoking a backtrace procedure and added to $RT(N_i)$. Otherwise, v_{min} is expanded and all its neighbors which have not been previously visited are inserted in the PQ. Maze expansion continues until paths to all sinks are found.

IV. Acceleration Strategies

Two main approaches for accelerating FPGA routing are

- routing multiple non-overlapping nets in parallel [8] and
- accelerating the maze expansion [8].

Given that our hardware platform is limited to a single CPU+FPGA and a single memory and that maze expansion accounts for approximately 68% of the total runtime [9], accelerating maze expansion is the straightforward decision. To enable the accelerator to work with large RRGs, we use the on-board memory and not the on-chip memory for RRG storage. Our acceleration strategies are (1) reducing the number of irregular memory accesses, (2) exploiting the available, albeit limited, parallelism in maze expansion, and (3) implement the priority queue in hardware to reduce the latency of the queue operations compared to the software queues. The following sections address these three strategies in detail.

A. Reducing Irregular Memory Accesses

Traversing RRG graph during maze expansion translates to making many irregular memory accesses. We propose two novel strategies for reducing their number:

- Allow wire-to-pin expansions *only* immediately before arriving to the sink node. As a consequence, many unnecessary memory accesses are avoided and what is left are mostly wire-to-wire expansions.
- Design RRG data layout that allows for efficient streaming memory accesses when performing wire-to-wire expansions.

The latter is possible because the regularity of FPGA architecture allows the wire segments to be grouped. We choose to group all the wires starting at the same (x,y) coordinate of the FPGA grid and going in the same direction (left, right, up, down) in the same *set*; this grouping can be done regardless of the wire segment length (1 or longer). Consequently, instead of loading from the memory a single wire (RRG node) to expand it to another wire, thus issuing two random memory access, one can request two burst accesses for loading two entire wire sets, and have them expanded in parallel by an FPGA accelerator. If all the wires in a set take part in the same wavefront, this approach effectively reduces the number of random memory accesses by a factor equal to the number of wires in a set (half of the routing channel width, for length-1 wires). However, if some of the wires do not have to be expanded, one spends extra time to perform unneeded data movement and processing. We accept to take this risk, as many other accelerators for graph traversal algorithms [3]–[6] accept to trade computation for streaming access to nodes and/or edges of a graph.

B. Parallel Computation

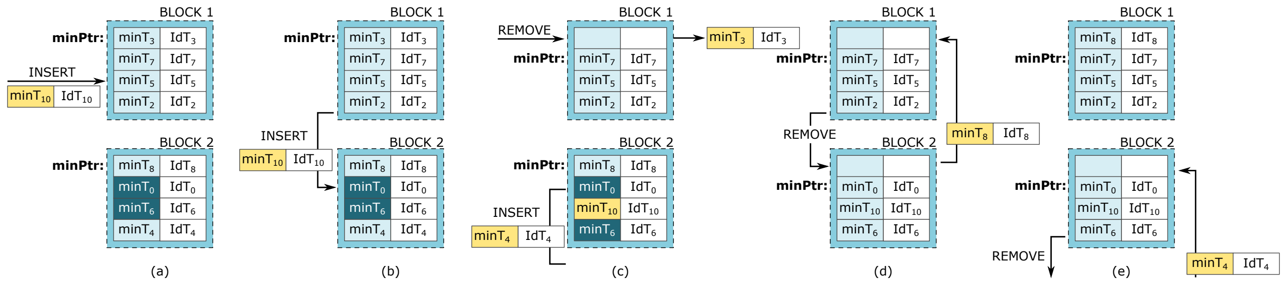
In maze expansion, parallelism opportunities are very hard to discover and manage, as conflicts occur whenever two RRG nodes attempt to expand to the same neighboring vertex. Our implementation, in which we group wires into sets, is particularly suitable for making full use of the limited available parallelism. Firstly, exploiting physical locality when grouping the FPGA wires provides good separation between potentially conflicting RRG nodes. Secondly, common routing switch topology (Universal, Wilton, Disjoint) with flexibility $F_s = 3$ do not allow two wires from the same set to drive the same wire in a different set. Hence, the amount of exploitable parallelism may even be equal to the size of the set, allowing our FPGA accelerator to expand all the wires in a set in parallel.

C. Efficient Priority Queue

Sorting priority queues in software is known to be a time-consuming operation; in VPR, the heap operations account for about 25% of the total routing time [10]. To improve the queue performance, we leverage on the inherent spatial computational properties of FPGAs and implement our priority queue in hardware.

The queue is implemented as a chain of identical blocks. These blocks operate in parallel; they can independently perform insertion and removal of an element from the queue, while maintaining it sorted. A similar approach was used by Rios et al. [10]. Let us illustrate our priority queue (pqueue) implementation on an example in which the chained blocks have only four elements each. Figure 1a) shows the first two blocks in one such queue. An element is composed of a pair of values, where the first is the minimum of all the total costs of the nodes in set T_i, while the second is the ID of set T_i. Every block has a pointer $minPtr$ for keeping the position of the minimum element in the queue.

Insertion of an element, for example $\{\min T_{10}, \mathrm{Id}T_{10}\}$ between elements $\{\min T_0, \mathrm{Id}T_0\}$ and $\{\min T_6, \mathrm{Id}T_6\}$, which reside in Block 2, happens as follows. Firstly, the value $\min T_{10}$ is compared to all the values in Block 1. Given that it is larger than them, the insert request is passed to the next block in the chain: Block 2 (Figure 1b). Comparing $\min T_{10}$ to the values



BLOCK 1
BLOCK 2
minPtr:
INSERT
REMOVE
minT_{10} IdT_{10}
minT_{3} IdT_{3}
minT_{8} IdT_{8}
minT_{4} IdT_{4}
(a)
(b)
(c)
(d)
(e)

Fig. 1: Priority queue illustration using two chained blocks of four elements.

in Block 2 reveals that the new element should be placed immediately after $\{\min T_0, \mathrm{Id}T_0\}$. Then, the elements below T_0 get shifted for one position and the highest among them is passed to the Block 3 along with the insert request.

Removal of an element from the queue is equivalent to returning the element with the min cost, whose position is kept in the pointer $minPtr$ (Figure 1c). The remove request is passed to the next block in chain, Block 2. As a result, the min cost element from Block 2 ($\{\min T_8, \mathrm{Id}T_8\}$) is returned to Block 1 and written in place of the removed element (Figure 1d). The remove request is passed to the Block 3, and so on until the last block in the chain (Figure 1e). At the same time, the pointers $minPtr$ in every block are updated to keep the position of the respective min elements.

This approach effectively hides the queue insertion and removal latency. However, the limitation of the on-chip queue is the amount of available on-chip memory, which depends on the chosen FPGA. Limited queue size can affect the algorithm convergence (the number of iterations before the routing completes), the quality of the final routing solution (more paths can be explored if the queue size would be unlimited), and the size of the design that is routable by the FPGA accelerator.

V. Our Maze Routing Algorithm

Our maze router builds on the Dijkstra's algorithm for shortest path search, except that, in our case, the wavefront is composed of wire sets. Routing of a sample source-sink pair is illustrated in Figure 2, while Figure 3 is the algorithm flowchart.

Routing begins by CPU expanding the net source to the CLB output pins (Figure 2a, in red, and Figure 3a). Then, the CPU expands the wavefront from the output pins to those wire sets that can be driven by them (Figure 2b). CPU then inserts to the hardware priority queue (pqueue) this initial wavefront; an element in the pqueue is a pair of values $\{minTi, IdTi\}$, where $IdIt$ is the ID of the set Ti and $minTi$ is the minimum cost of all the nodes in the set that were touched during the expansion from the output pins to Ti. Additionally, the CPU marks as *targets* all the wire sets that can provide direct connection to the input pins of the sink (Figure 2, in blue). As long as the sink is not reached, the algorithm repeats the following procedure. The CPU removes the minimum cost set from the pqueue; we call this set a *start set*. If the start set is inside the net's bounding box and it is a wire set, the CPU reads its adjacency list to discover the neighboring sets; we call them the *end sets*. Then, the CPU issues the following commands to the FPGA accelerator: (1) load the nodes in the start set, (2) load the outgoing edges of the start set, and (3) load the end sets that are within the net's bounding box (Figure 3b). The FPGA accelerator responds to the commands by autonomously performing wavefront expansion (Figure 2c), inserting in the pqueue new elements, and storing the updated sets back to memory (Figure 3c). Once the FPGA is done expanding the start set, the CPU takes over and removes the next start set from the pqueue. If the new start set is a target (Figure 2, in blue), i.e. it may lead to the sink, the software expands it to uncover all the input pins and to insert them to the pqueue (Figure 3d). If the new start set is one of the input pins leading to the sink (Figures 3e and 2e), the CPU builds the partial routing tree, updates present congestion costs, cleans the pqueue and re-initialize it with the partial routing tree (at zero cost), while issuing a command to the FPGA to reset the RRG nodes, as preparation for routing the next sink-source pair. This algorithm produces deterministic results.

VI. Implementation

Our hybrid CPU+FPGA system consists of a dual core processor, customized FPGA modules and an external DRAM memory (Figure 4). The input files (RRG and placed circuit netlist) and output files (net routing trees) are stored on an SD card with a Ubuntu Core 14.04 Linux distribution (a lightweight distribution commonly used in embedded systems). All FPGA hardware modules are developed as Register Transfer Level models in VHDL. The key HW modules are

- wavefront expansion accelerator,
- priority queue, implemented following the design presented in Section IV-C, and
- reset module, a custom DMA unit for burst access to the RRG data and resetting the node costs, as part of the preparation for routing the next source-sink pair or the next net.

The FPGA and the CPU communicate via a command FIFO queue, allowing them to operate at their own paces and requiring least synchronization. The main system memory is shared between FPGA and CPU. The DRAM address space used for data storage is reserved during the system's boot process, preventing the kernel from accessing it. Data is accessed from

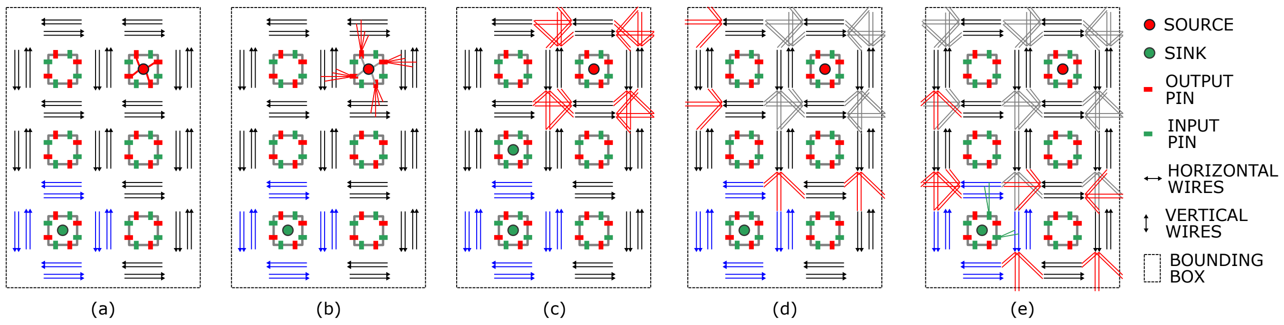


Fig. 2: Wavefront expansion, hop by hop.

the user space by memory mapping the data region in the user application. RRG data is not cached, yet this does not create performance penalties as it is the FPGA and not the CPU that accesses it most of the time. High-speed FPGA-to-SDRAM ports are used for transfers from/to memory. These are significantly faster than HPS-to-FPGA bridges for communication between ARM and FPGA, which pass through interconnect logic. The lightweight HPS-to-FPGA bridge is used for issuing commands to FPGA, whereas the high performance HPS-to-FPGA bridge is used for inserting and removing elements from the hardware pqueue.

A. Routing Resource Data in Memory

Routing resource data consists of RRG nodes, RRG edges, and adjacency lists. RRG nodes and edges are stored in dedicated memory buffers located in the address space shared by the FPGA and the CPU; this space is not cached. Adjacency lists, however, since they are accessed only by the CPU, are stored in the address space which is accessible by CPU only and which is cached.

An RRG node is represented with two data structures. The first contains three 32-bit fields: total path cost (minimum of the costs of all the discovered paths from the net source to this node), total node cost, and the identification number (ID) of the predecessor node on the minimum-cost path (Figure 5). This structure is stored as part of the *node data buffer* in the shared data region. The second data structure contains two 32-bit fields: node occupancy and the accumulated cost. Since these are updated only by the CPU, this structure is stored in the cached data region.

All RRG nodes that represent FPGA wires are split into sets, where a set groups all wires starting at the same (x, y) FPGA coordinate and going in the same direction. These and other RRG nodes are stored in the following order: source nodes first, then output pin nodes, node sets, input pin nodes, and finally sink nodes (Figure 5).

Adjacency list describing connections between source nodes and output pins, output pins and wires, wires and input pins, and input pins and sink nodes, are stored immediately after all RRG nodes.

Connections between sets are represented as adjacency lists as well; for every start set and all of its end sets, the end set ID and the total number of edges from start set to the end set are saved. Edges between the nodes in two connected sets are stored in the *edge list*. Each outgoing edge between a node in one set and a node in another is described with 8 bits for the node offset in the start set and 8 bits for the node offset in the end set. Outgoing edges of a set are stored sequentially in the edge data buffer (edge list), making it possible to stream them from the external memory. Edges between sources and pins, pins and FPGA wires, pins and sinks, as well as connections between sets, are kept in the cached memory space, as it is CPU that accesses these data during routing.

The time required to create these data structures is negligible compared to the time to route a net. Most importantly, this data is created only once for a specific FPGA architecture.

B. Wavefront Expansion Accelerator

The block diagram of the wavefront expansion accelerator is shown in Figure 6. This unit interfaces the CPU via the command FIFOs. Three commands are supported: `LOAD START_SET`, `LOAD END_SET`$_i$, and `LOAD OUTGOING_EDGES(START_SET)`. Each command is composed of a descriptor, the starting address, and the length of data to fetch to one of the custom DMA units integrated in the accelerator.

Two DMAs are used for node operations: one for loading and the other for storing the nodes to memory. Both DMAs can load or store one node per cycle (for our design running at 150 MHz, DMA bus size of 128b, the required bandwidth of these DMA units is 2.4 GBps). The third DMA is responsible for loading edges between two sets. All DMAs use burst memory accesses.

The accelerator continuously polls the command FIFOs to see if there are any pending requests from the CPU. As soon as a command is available, it is read and its execution initiated. We implement two separate FIFOs for node- and edge-loading commands, to allow the accelerator to issue memory requests in parallel. As soon as the nodes of the start set and the nodes of the first end set (END_SET_1) are loaded to the local BRAMs, the controller starts streaming the edges between the two sets and performs wavefront expansion. In parallel, END_SET_2 is loaded to internal memory. Once the loading is done, the expansion of END_SET_2 starts; in parallel, once

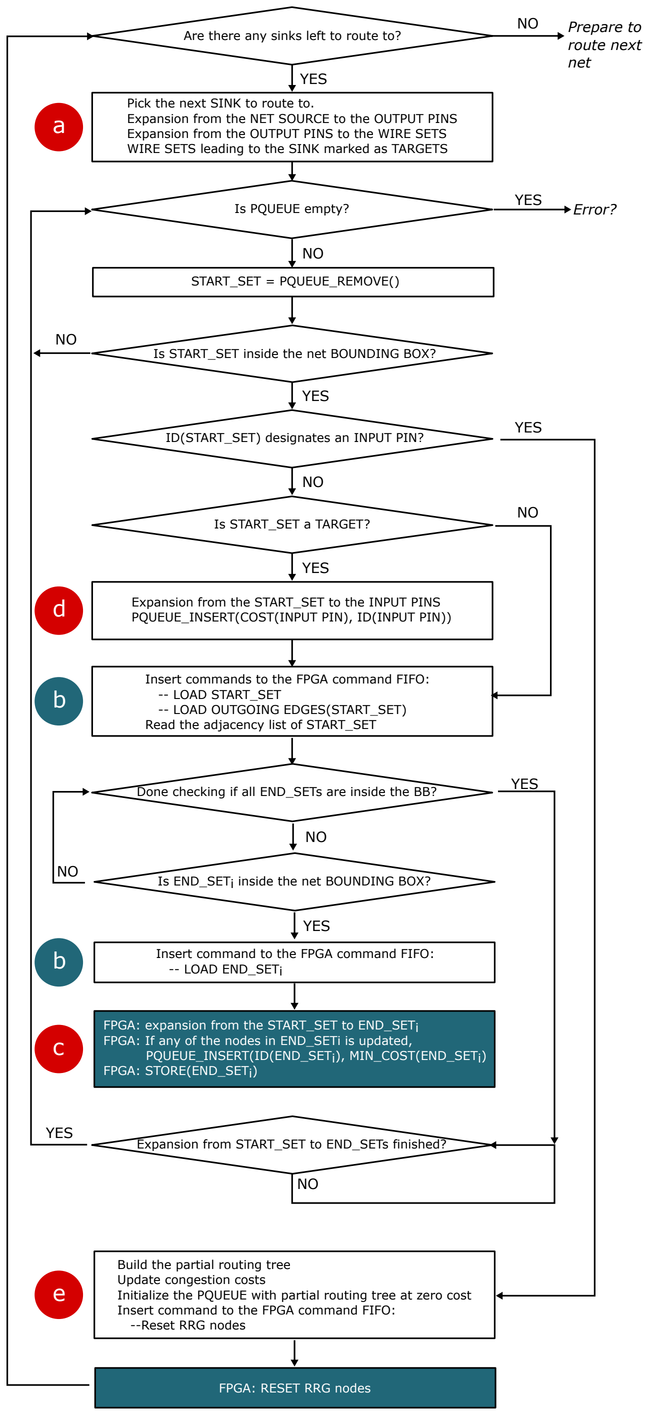

Are there any sinks left to route to?
NO
Prepare to route next net
YES
a
Pick the next SINK to route to.
Expansion from the NET SOURCE to the OUTPUT PINS
Expansion from the OUTPUT PINS to the WIRE SETS
WIRE SETS leading to the SINK marked as TARGETS
Is PQUEUE empty?
YES
Error?
NO
START_SET = PQUEUE_REMOVE()
NO
Is START_SET inside the net BOUNDING BOX?
YES
ID(START_SET) designates an INPUT PIN?
YES
NO
Is START_SET a TARGET?
NO
YES
d
Expansion from the START_SET to the INPUT PINS
PQUEUE_INSERT(COST(INPUT PIN), ID(INPUT PIN))
b
Insert commands to the FPGA command FIFO:
-- LOAD START_SET
-- LOAD OUTGOING EDGES(START_SET)
Read the adjacency list of START_SET
Done checking if all END_SETs are inside the BB?
YES
NO
NO
Is END_SETi inside the net BOUNDING BOX?
YES
b
Insert command to the FPGA command FIFO:
-- LOAD END_SETi
c
FPGA: expansion from the START_SET to END_SETi
FPGA: If any of the nodes in END_SETi is updated,
PQUEUE_INSERT(ID(END_SETi), MIN_COST(END_SETi))
FPGA: STORE(END_SETi)
YES
Expansion from START_SET to END_SETs finished?
NO
e
Build the partial routing tree
Update congestion costs
Initialize the PQUEUE with partial routing tree at zero cost
Insert command to the FPGA command FIFO:
--Reset RRG nodes
FPGA: RESET RRG nodes

Fig. 3: Our maze router based on Dijkstra's algorithm.

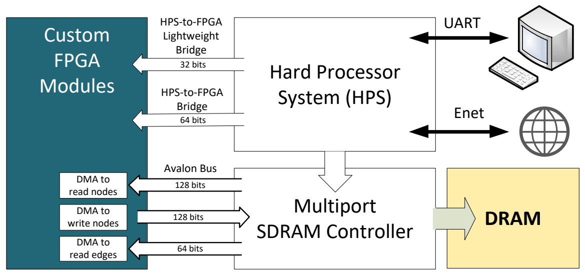


Fig. 4: System block diagram.

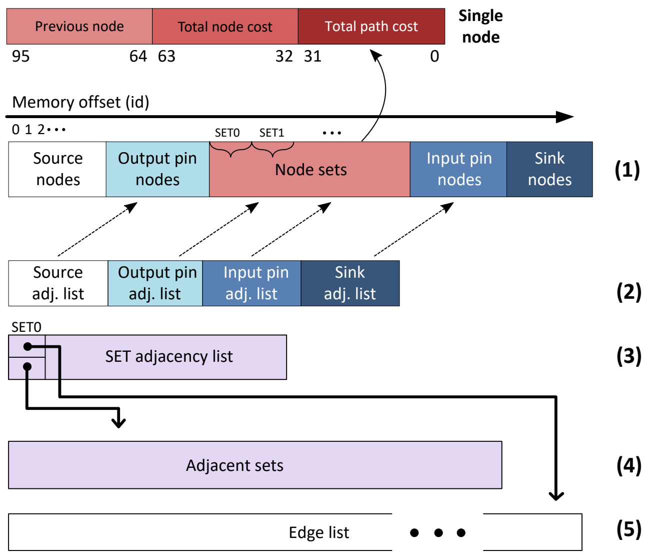


Fig. 5: Routing-resource data structures. (1) RRG nodes. (2) Node adjacency list for sources, sinks, and pins. (3) Node sets. (4) Set adjacency list. (5) List of edges between wires in two neighboring sets.

the expansion of the END_SET_1 is done, END_SET_1 is stored back to memory.

The controller is responsible for keeping track of the minimum cost of all the nodes that were updated during the expansion of one set and to insert this value to the pqueue along with the ID of the corresponding set. The main components of the processing kernel are the BRAMs, the FP adder, and the FP comparator (Figure 7). Multiple processing kernels are instantiated, to allow parallel edge processing. Each kernel contains one BRAM for start set nodes, and three BRAMs for three different end sets. This makes it possible to pipeline the operations—while one BRAM is loaded, the other can be expanded (read, modified) and the third can be stored back to memory). The number of processing kernels is a generic parameter in the design and can be changed prior to compilation. The only limitation is that it needs to be a power of two. We chose to instantiate four of them, as in our implementation four edges could be fetched in one cycle.

All operations on nodes are performed on floating point (FP) numbers with single precision. FP units are imported from the Alteras IP library of components. To improve the design frequency, the latency of the FP adder is set to 14 cycles, the latency of the FP comparator is set to 3 cycles, and pipeline registers are added.

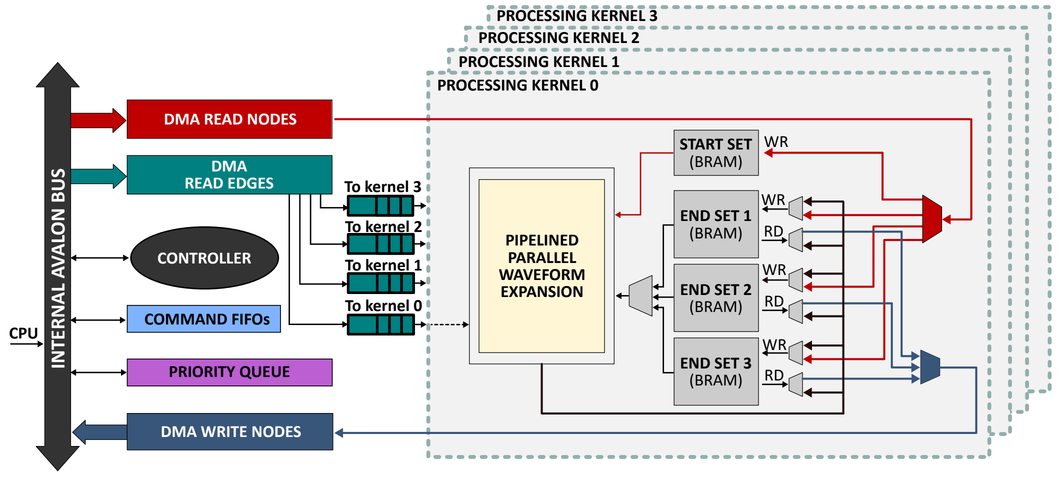


Fig. 6: Wavefront expansion accelerator. The hardware priority queue, the accelerator, and the CPU are all connected to the internal Avalon bus, which handles all the arbitration.

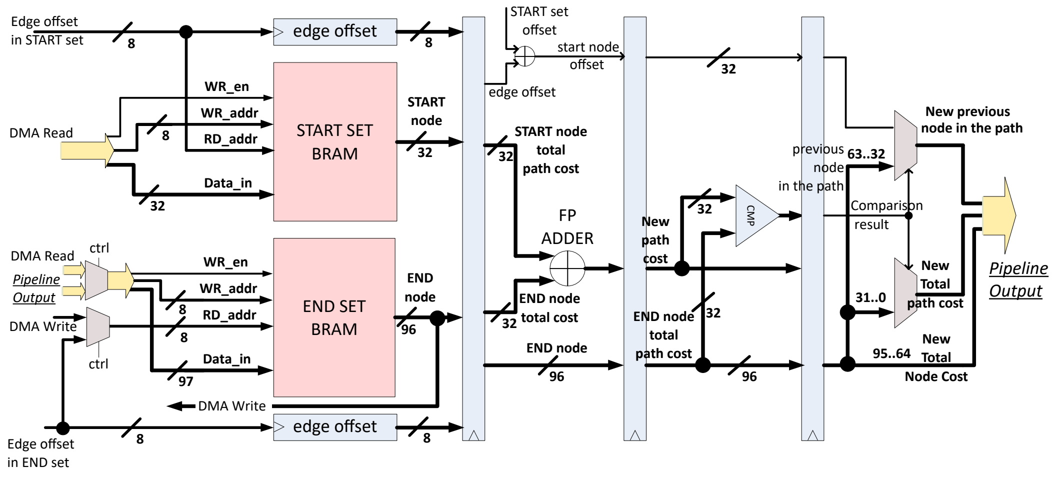


Fig. 7: Processing kernel pipeline illustration. There are four BRAMs, one for start set and three for end set. For simplicity, only one BRAM for one end set is shown; other two are used in the same way. Similarly, not all pipeline registers are shown.

For a common FPGA routing switch topology with flexibility $F_S = 3$, no two nodes from one set can connect to the same node in the neighboring set and, consequently, no data dependency checking nor data forwarding is required. Additionally, only one processing kernel will have the correct updated costs of an end set. To find it, we tag an end set node with a dirty-bit to indicate that this node was updated. These flag bits are also used as decoder selection signals to pass the correct values to the output.

VII. Experimental Evaluation

Our design is implemented on DE1-SoC, a mid-end CPU+FPGA platform by Intel. This platform was chosen because it has everything needed to prove the concept behind the proposed hardware-accelerated router and because the interface between the CPU and the FPGA is relatively simple, compared to more powerful CPU+FPGA platforms. The system contains a Cyclone V SoC-FPGA with dual core ARM Cortex A9 processor. Each core is running at a 925 MHz clock frequency. A 32 KB L1 cache and a 512 KB L2 cache are available. The external memory is 1GB DDR3 operating at 400 MHz, providing 25 Gbps of bandwidth at 100% of efficiency. We measured the maximum bandwidth of 14.4 Gbps. To run the VPR router, we use a PC with Intel Core-i5 dual-core CPU having 4 logical processors running at 2.2 GHz and 3 MB of cache.

We evaluated the performance of our router on a subset of circuits in the VTR benchmark suite [11] and two FPGA architectures:

- *Arch-k4-l1*: k4_N4_90nm at length-1 wire segments and
- *Arch-k6-l4*: k6_frac_N10_chain_mem32K_40nm and length-4 wire segments.

For every benchmark, we use Odin II for logic synthesis, ABC for logic optimization and technology mapping, and VPR for

TABLE I: Resource usage.

Resource	Available	Used
ALMs	32,070	31,374 (98%)
Registers	128,300	46,850 (37%)
Block memory bits	4,065,280	1,725,904 (42%)
RAM blocks	397	389 (98%)
PLLs	6	1 (17%)
Pins	457	132 (29%)

pack and place. Routing is done using a benchmark-specific channel width, equal to 1.3× the corresponding minimum channel width reported by VPR router, and bounding box factor set to 3 for Arch-k4-l1 and 4 for Arch-k6-l4. Our preprocessing script is embedded in the VPR source code; it outputs a binary file containing the routing resource data and the circuit netlist. This file is saved on the board's SD card, at which point the acceleration platform is ready. Once the routing is finished, a binary file with final routing trees is produced. We import it back to VPR, where it is converted to its .route format, allowing us to use VPR's timing analysis. All the experiments use VTR 8.0 development trunk [12].

A. Resource Usage and Design Frequency

Table I shows the usage of FPGA resources. The most resource consuming design module is the pqueue, for two reasons. Firstly, floating point comparisons are performed when an element is to be inserted. Secondly, the pqueue should be as deep as possible to keep as many sets as possible and improve the algorithm convergence.

As an example, the largest benchmark that we tested (mkSMAdapter4B) did not route successfully with the priority queue of 16K elements, where the elements were distributed in chained BRAMs each holding 128 elements. When the queue depth was increased to 20K elements, the benchmark routed successfully. Alternative approach to increasing the queue depth would be to increase the size of every block in the chain, i.e. to have the BRAMs keep 256 elements instead of 128. However, this would have a negative impact on the router performance, because the insertion of elements into the queue would then take more time. To improve convergence, we chose to organize the pqueue in blocks of 128 elements each and to limit the queue depth to 20k elements. As a consequence, almost all the available on-chip memory is in use.

The maximum achieved design frequency is 160.72 MHz, but we run the experiments at 150 MHz.

B. Experimental Results

To evaluate the efficiency of our accelerator, we measure the time to route various benchmarks in several different configurations: (1) our router on ARM+FPGA, (2) our router on ARM only, (3) VPR router on an Intel Core-i5 CPU and (4) our router on the same Core-i5 CPU. Purely software routers are all compiled with -o5 flag.

Figures 8 and 9 show the speedup of all these routers. The baseline is our router running on ARM. In comparison, our accelerated router on ARM+FPGA is 1.91–3.39× faster for Arch-k4-l1 architecture and 1.23–2.13× faster for Arch-k6-l4

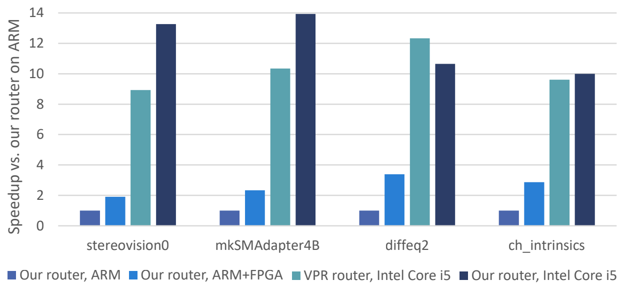


Fig. 8: Speedup relative to our router running on embedded ARM processor and *Arch-k4-l1* architecture.

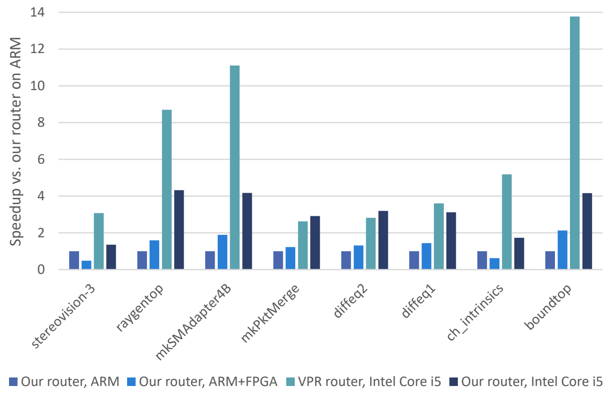


Fig. 9: Speedup relative to our router running on embedded ARM processor and *Arch-k6-l4* architecture.

(exceptions are stereovision-3 and ch_intrinsics benchmarks, which, being very small circuits, permit ARM cache memories to gain advantage over FPGA's burst memory accesses). Compared to our router running on the ARM+FPGA, VPR running on a top notch Intel Core-i5 CPU is still faster: 3.36–4.69× for Arch-k4-l1 (on average 4.03×), and 2.13–6.46× for Arch-k6-l4 (on average 4.42×). However, our router running on the same commercial-grade CPU sometimes outperforms VPR; whether it is faster or slower depends on the number and the size of sets: for Arch-k4-l1 architecture it is often faster, while for Arch-k6-l4 it is often slower. Therefore, our strategy for grouping wires in sets may help improving router speed in software as well.

We also compare the quality of the routing output with that of VPR. On average, our router reports less than 0.5% increase in total wirelength, average net length, and critical path compared to VPR router.

C. Discussion and Performance Predictions

The factors influencing the performance of our design are the following: (1) external memory bandwidth, (2) CPU speed, (3) the fact that CPU and FPGA are competing for the same data storage, and (4) the maximum accelerator frequency, which is limited by floating-point operations.

Although our accelerator issues memory read and write requests in parallel, given that the system has one single-port DDR memory, these requests are, eventually, processed sequentially. Therefore, the accelerator cannot achieve its full potential; in Section VI-B we show that the processing kernel has three local BRAMs for end sets. Ideally, this would allow loading one set, accessing the second (for wavefront expansion), and storing the third set to DRAM to be executed in parallel. To observe what actually happens, we use Chipscope to measure the time for expanding a set; it results to be higher or equal to

$$T = t_{R\_START} + t_{R\_EDGES} + \sum_{i=1}^{N} (t_{R\_END_i} + t_{EXP} + t_{W\_END_i}), \tag{1}$$

where R stands for reading, W for writing, and N is the number of adjacent end sets. Surprisingly, loading a new set is done after storing the previous. This is because the ARM takes more time to test if a set should be loaded (bounding box check) and to issue a command to the FPGA than what the FPGA takes to expand a set. A different platform, equipped with a faster CPU and a faster CPU-to-FPGA link, would prevent this issue from happening.

Since our accelerator may have a different performance if ported to another platform, let us attempt to estimate possible improvements; of course, exact estimates can only be obtained after thorough experimental testing. For simplicity, let us assume that all memory accesses to sets and edges take the same time t_{SET}, and that $N = 3$ (the case for Arch-k4-l1). Hence, Eq. 1 becomes

$$T_{DE1-SoC} = 8 \times t_{SET,DE1-SoC} + 3 \times t_{EXP,DE1-SoC}. \tag{2}$$

Then, let us imagine a platform that offers higher memory bandwidth, faster CPU, and larger and faster FPGA, such as Intel HARP [13], featuring Intel Xeon CPU, Stratix V FPGA, and a DDR3 memory operating at 1600 MHz. Since HARP's memory provides $4\times$ higher bandwidth and Stratix V provides $1.6\times$ faster floating-point units [14], time to expand a set would be reduced to

$$T_{HARP} \cong 2 \times t_{SET,DE1-SoC} + 1.875 \times t_{EXP,DE1-SoC}. \tag{3}$$

Under the worst case assumption that $t_{EXP} = t_{SET}$, T_{HARP} would be about $3\times$ shorter than T. If the system memory would be true dual port, expansion could be done in parallel with loading new and storing old sets and T could be approximated by

$$T' = 3 \times t_{SET} + 3 \times t_{EXP} \cong 6 \times t_{SET} \tag{4}$$

For only $4\times$ higher memory bandwidth, the total speedup could be thus increased to $\cong$ 7.33, and our accelerator would outperform VPR. Hence, in view of ever improving performance of platforms that couple CPUs with FPGAs, we believe that the solution we propose may only gain in performance compared to the software router.

VIII. Conclusions

In this paper, we propose three acceleration strategies for an FPGA router ported from software to a CPU+FPGA hardware platform. We design and implement the router, and run a set of benchmarks from the VTR suite [11] to compare its performance to VPR. The results show that our implementation produces deterministic and good quality output. It is also successful in accelerating a purely software version on the same CPU+FPGA platform, but not against a powerful Intel Core-i5 CPU, due to the limitations of the mid-end DE1-SoC acceleration platform. The performance prediction model we provide suggests that more memory bandwidth and faster FP units would render this FPGA-assisted router superior to the software alternative.

References

[1] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: Architecture and CAD for FPGAs from verilog to routing," in *Proc. of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2012, pp. 77–86.

[2] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proc. of the 3th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 1995, pp. 111–17.

[3] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *IEEE 24th Annual Intl. Symp. on Field-Prog. Custom Computing Machines*, Washington, DC, Aug. 2016, pp. 103–110.

[4] ——, "Accelerating large-scale single-source shortest path on FPGA," in *IEEE International Parallel and Distributed Processing Symposium Workshops*, Hyderabad, India, May 2015, pp. 129–136.

[5] T. J. Ham, L. Wu, N. Sundaramz, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Lanzarote, Spain, Dec. 2016, pp. 1–13.

[6] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph processing framework on FPGA: A case study of breadth-first search," in *In Intl. Conf. on Field-Programmable Gate Arrays (FPGA), ACM.*, Monterey, CA, 2016, pp. 105–110.

[7] A. DeHon, R. Huang, and J. Wawrzynek, "Hardware-assisted fast routing," in *Proc. of the 10th IEEE Symp. on Field-Programmable Custom Computing Machines*, Boulder, CO, Apr. 2002, pp. 1–11.

[8] M. Stojilović, "Parallel FPGA routing: Survey and challenges," in *Proc. of the 27th Intl. Conf. on Field-Progr. Logic and Appl.*, Ghent, Belgium, Sep. 2017, pp. 1–8.

[9] M. Gort and J. H. Anderson, "Accelerating FPGA routing through parallelization and engineering enhancements," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 61–74, Jan. 2012.

[10] J. Rios, "An efficient FPGA priority queue implementation with application to the routing problem," pp. 1–11, 2007.

[11] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, pp. 6:1–6:30, Jun. 2014.

[12] VTR, "ff58f9791," 2018. [Online]. Available: github.com/verilog-to-routing/vtr-verilog-to-routing

[13] Intel, "Ivytown xeon + FPGA: The HARP program," 2016. [Online]. Available: https://cpufpga.files.wordpress.com

[14] ——, "Floating-point IP cores user guide," 2016. [Online]. Available: https://www.intel.com