LegUp-NoC: High-Level Synthesis of Loops with Indirect Addressing

Asif Islam
University of Waterloo
Ontario, Canada
a27islam@uwaterloo.ca

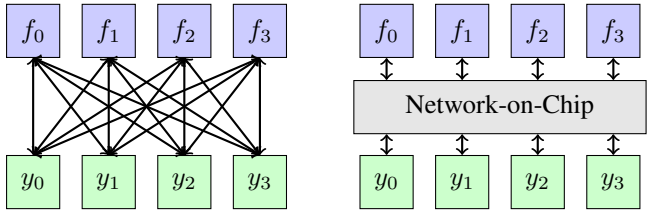
Nachiket Kapre
University of Waterloo
Ontario, Canada
nachiket@uwaterloo.ca

***Abstract*—**

Loops with indirect addressing, of the type `A[B[i]]`, are notoriously difficult to parallelize using contemporary FPGA High-Level Synthesis (HLS) tools. In contrast, loops with direct addressing can be parallelized using compile-time approaches by replicating datapaths and memory blocks. Such compile-time approaches do not work for indirect addressing as indices `B[i]` are not known until runtime. Consequently, since addresses may point to any memory bank, HLS tools generate expensive cross-bars between datapaths and memory banks. As all datapaths may target the same bank in a given cycle, a sequential arbitration is provided to control the crossbar multiplexers. In this paper, we show how to overcome the resource and performance limitations of existing tools using a Network-on-Chip (NoC) approach to route indirect indices to the memory banks over a packet-switched fabric. NoCs provide scalable connectivity between FPGA datapaths and memory banks and allow parallel routing of packets from datapaths to the banks. We develop a LegUp 5.0 compiler pass that (1) handles loops with indirect memory access by inserting NoCs into the circuit as required, (2) provides a performance and resource tuning framework for optimizing the resulting hardware, and (3) obviates the need for NoC expertise during programming. We quantify the effectiveness of our approach across a range of kernels with indirect accesses by comparing against baseline LegUp 5.0 targeting a Xilinx VC707 board. For synthetic indexing at 256 threads, we observe an improvement of 150× LUTs, 4–5× Fmax, 15–16× II for `UNIFORM RANDOM` indexing. For real-world case studies such as Sparse Matrix-Vector multiplication, Graph Analytics and 1-D FFT, we see 5–20× speedups for 16–256 threads with a 20–30% overhead for adding the NoC infrastructure.

I. Introduction

FPGAs are now first-class computing devices that enjoy increasing adoption in data-centers such as Microsoft Azure, Amazon F1, Baidu, Huawei, and Alibaba clouds. RTL design of data-center applications using VHDL/Verilog is ill-suited for these cloud-based FPGA platforms due the constantly changing nature of web workloads and the diversity of cloud hardware platform configurations. High-level programming approaches using OpenCL or C/C++ supported by HLS (High-Level Synthesis) compilers make it possible to keep pace with the rapid evolution of the software codebase while providing platform portability. Using hardware-aware HLS programming patterns and appropriate use of compiler directives, it becomes possible to automatically generate RTL that can close the quality gap with hand-optimized code.



(a) LegUp Crossbar (b) NoC Interconnect

Fig. 1: For loops with indirect addressing, FPGA HLS tools generate crossbars with round-robin scheduling between datapaths (f_i) and memory banks (y_i) for loops with indirect addressing (left). NoCs can enhance this interconnect with a superior alternative (right).

HLS tools like LegUp [3] and Vivado HLS are aimed primarily at parallelization of loop-oriented code with additional support for task parallelism and dataflow parallelism. Loops can be optimized using a wealth of techniques such as loop pipelining, loop unrolling, loop tiling, and loop partitioning. This is possible when loops have memory operations based on direct addressing where memory addresses are simple linear functions of the loop index. This allows the generated hardware structures, the datapath and attached RAM memory blocks, to be neatly partitioned and replicated in a scalable manner. Irregular addressing is found in important computational kernels of contemporary interest such as deep learning (sparse matrix algebra), graph analytics (sparse graph algorithms), databases (sorting), signal-processing (FFTs), and other areas. HLS tools currently do not support indirect addressing in an effective manner due to the complex interaction between the datapaths and memory banks (Figure 1a). In this paper, we address this important limitation of FPGA HLS tools by using packet-switched networks.

FPGA overlay NoCs (Network-on-Chips) are used for connecting hardware components using a packet-switched style of communication and a resource-shared hardware infrastructure. In the extreme case, it is possible to connect the different N hardware components using a crossbar for high-performance at high-cost $O(N^2)$ or shared busses/rings for low-performance at low-cost $O(1)$ solutions. NoCs offer a resource-efficient, high-performance alternative that enables sharing of hardware while permitting dynamic scheduling (routing) of packets in

```
#pragma omp parallel for num_threads(UNROLL)
for(int i=0;i<N;i++) {
  // direct addressing
  y[i]=f(x[i]);
}
```

Fig. 2: Simple loop with direct addressing, array indices for x and y arrays are the loop iterator i. Trivial to parallelize such loops through existing HLS tools and pragmas.

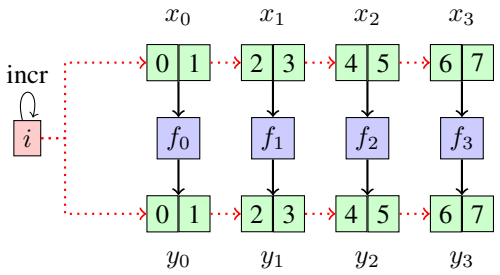


Fig. 3: Parallelizing loop with direct addressing using unrolling + array partitioning in LegUp.

the fabric. In this paper, we show how to deploy NoCs in the HLS toolflow without programmer intervention to automatically parallelize loops with indirect indexing (Figure 1b).

We make the following key contributions in this paper:

- We quantify the cost-performance trends for loops with irregular indexing when using LegUp 5.0 HLS tool.
- We develop a LegUp 5.0 compiler pass that inserts NoCs between HLS-generated datapaths and memory banks.
- We evaluate the cost-performance benefits of NoC-assisted loop parallelization across a range of irregular benchmarks, and case studies involving Sparse Matrix-Vector Multiplication, Graph Analytics, and FFT.

II. Background

FPGA HLS tools generate hardware that balances area usage with performance to make the most efficient use of available resources. Performance is measured in terms of two metrics: (1) Latency: number of cycles required to evaluate the loop, and (2) Initiation Interval (II), or the number of cycles between consecutive loop iterations in hardware.

A. Loop Parallelization for Direct Loops

For simple loops with known bounds, and direct indexing, the use of loop pipelining optimizations will allow launching of a new loop iteration each cycle (II=1) at high frequencies (high latency). If FPGA developers are willing to spend FPGA resources, they can improve performance through loop unrolling optimizations. Here, the compiler creates identical copies of the loop hardware and divides the iterations across the copies. In Figure 2, we show a simple for-loop with direct addressing of x and y arrays based on loop index i. We can visualize the hardware implementation of this loop in Figure 3. The `omp parallel for` OpenMP pragma hint is used by the HLS compiler to generate multiple copies of the function f along with partitioned memory banks of the arrays x and y. This is possible only if (1) multiple copies of f and memory

```
#pragma omp parallel for num_threads(UNROLL)
for(int i=0; i<N; i++) {
  // indirection through dest[]
  y[dest[i]] = f(x[i]);
}
```

Fig. 4: Loop with indirect addressing, array access to y requires an indirection through $dest$ array. Difficult to parallelize due to unknown contents of $dest$.

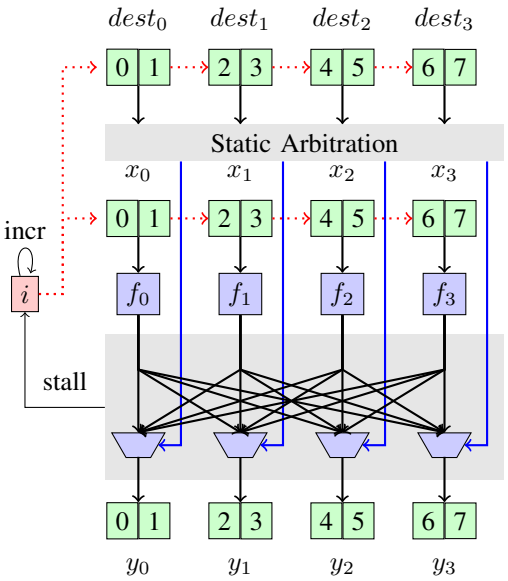


Fig. 5: Parallelization of loop with indirect addressing. Expensive crossbar needed as f can write to any memory bank. LegUp and Vivado HLS arbiters are static and sequential as every f can target same bank in same cycle.

banks can fit the FPGA device, and (2) array indices into the memory banks are strictly local to each copy. For $f = x^2$, we tabulate the results of the compilation in LegUp in Table I. We observe that II=1 for all cases, latency reduces smoothly with the unroll factor, while LUT cost increases as expected.

B. Loop Parallelization for Indirect Loops

Loops with indirect addressing are challenging to implement in hardware using HLS tools. (see Section V-A for discussion on related work in this area). In Figure 4, we show a simple example of a loop with indirect addressing where the indices into y array require an indirection through $dest$. Since the contents of the $dest$ array are unknown at compile time, it is not possible to parallelize the loops into independent copies in the same manner as the direct loops. We show the hardware generated by LegUp 5.0 and Vivado HLS to support the unrolled loop in Figure 5 and tabulate mapping results in Table I. An all-to-all multiplexer arrangement is necessary to allow the output of the many copies of f to write to any memory bank y. This structure is expensive as the multiplexer cost grows quadratically with the number of parallel instances of f and memory banks. To make matters worse, both LegUp and Vivado HLS use static sequential arbitration (1) to ensure bank conflicts can be avoided, and

TABLE I: Loop Parallelization of direct and indirect loops using LegUp 5.0 and Vivado 2017.3 for N=1024 iterations.

Unroll →	1	2	4	8	16	32
			Simple loop in Figure 2			
LUTs	84	446	943	1.9K	3.8K	8.2K
Clk (ns)	6.6	6.6	6.8	7.5	6.7	6.0
Cycles	32770	20491	10255	8210	8207	8207
			Indirect loop in Figure 4			
LUTs	209	696	2.4K	10.4K	41.4K	201.2K
Clk (ns)	6.6	7.1	7.8	8.7	15.9	18.8
Cycles	32770	32770	32770	32770	32770	32770

(2) (possibly) to vainly enforce sequential ordering where none is required from OpenMP's weak-consistency [6] semantics[1]. This slices up the timeslots on the memory bank ports resulting in large values of II that scale with the unroll factors. This defeats the entire purpose of parallelizing these loops and consumes quadratic FPGA area cost for no improvement in performance (a lose-lose situation).

C. FPGA Overlay NoCs

Loops with indirect addressing are expensive to realize on FPGAs with current HLS tools as we need wide-multiplexers to allow indirect access and arbiters for each memory bank to resolve potential conflicts in bank accesses and guarantee sequential consistency. Complexity of these hardware structures scales quadratically with loop unroll factor; for N banks and N datapaths, we need to implement N separate $N:1$ multiplexers. In contrast, FPGA overlay NoCs are implemented as a series of N router stages with small multiplexers per stage resulting in linear increase in cost. Instead of centralized static sequential scheduling used by HLS-generated arbiters, a NoC routes data dynamically using address information at runtime. Furthermore, we can customize the FPGA overlay NoC in a manner that allows the configuration to suit loop requirements. However, the use of a NoC requires explicit packetization of data, a different interface and handshake design, and awareness of deadlock considerations. Each NoC packet needs to be stamped with both the destination address, and the payload data. The destination address encodes the target memory bank and is used by the NoC to switch the packet to its intended destination. One must also worry about deadlock and livelock avoidance that can occur for certain indirect access patterns. This can be a tedious overhead for a software developer and require NoC design and usage expertise.

These considerations motivate the need for an automated HLS flow that can determine the effectiveness of how and where NoCs can be inserted in the generated hardware. We choose the Hoplite [9] NoC for this paper as it is an FPGA-optimized, low-cost implementation, but any cheap NoC will suffice. Hoplite routes single-flit packets (header + payload routed as one unit). The key to area efficiency of Hoplite is the use of deflection routing (no buffering cost) and choice of unidirectional torus topology (low-complexity router multiplexers). Unlike static scheduling that guarantees conflict-free

[1]Unless atomic operations are explicitly used

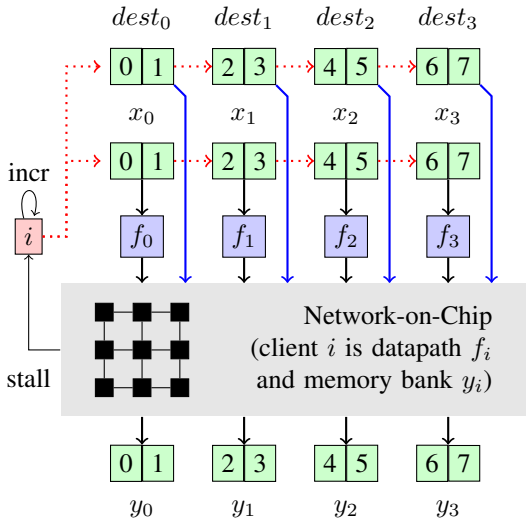


Fig. 6: NoC-assisted implementation of indirect addressing. Each NoC client i at position x, y has datapath f_i (injection) and memory bank y_i (ejection).

bank access with pessimistic assumptions, deflection routing manages conflicts through deflections. It remains to be seen if deflections can outperform static scheduling, and is the focus of our experimental evaluation.

III. LegUp-NoC

The key idea introduced in this paper is the use of FPGA overlay NoCs as a building block in High-Level Synthesis to support scalable parallelization of loops with indirect indexing. In this section, we show how to relate NoC behavior with HLS kernel performance, and describe how to integrate NoCs with HLS optimizations for two classes of indirect access patterns.

A. Interfacing with the NoC

In Figure 6, we show a high-level view of the interconnect between HLS-generated FPGA datapaths and partitioned memory banks for the code example with indirect addressing shown earlier in Figure 5. Here, the NoC routes *packetized* address and data tuples from $dest$ and the output of f respectively. The most significant bits of the index stored in $dest$ is used as the remote packet address (or lane address) for routing on the NoC. The local address into the memory bank and the data are sent as the payload of the packet. This packet format is shown below in Figure 7 and automatically constructed by the NoC-Datapath interface logic generated by our compiler pass.

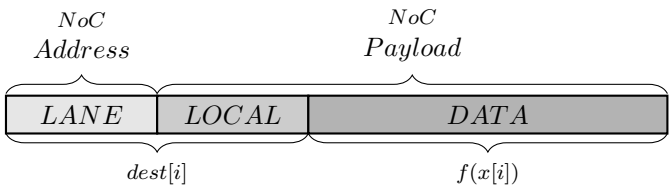


Fig. 7: Constructing a NoC packet for routing address + data for parallel loops with indirect addressing

Unlike existing HLS tools that use *static* round robin arbitration, we use the NoC to implement *dynamic*, runtime arbitration for us. Due to data-dependent arbitration, the path taken by the packets through the NoC and the resulting latency are known at runtime. Static arbitration requires pessimistic assumptions about the possibility of bank conflicts, and results in latency and initiation interval counts that are large. For dynamic arbitration, the properties of the indexing pattern is used to make better-informed, dynamic, runtime decisions.

We can understand the exact behavior of the NoC-assisted design in Figure 6, through an example. In this case, the second lane of the unrolled design will loop through the $2 \leq i \leq 3$. Let us assume that $dest[2] = 6$ and $dest[3] = 3$. The lane will dispatch two NoC packets with payloads $f(x[2])$ and $f(x[3])$. These packets will carry destination addresses 6 and 2 respectively. The NoC will bitslice the value read from $dest[]$ to compute the destination lane index (2 most-significant bits) and local address into the y memory bank (one remaining least-significant bit). Here, we will have $lane_{index} * 2^{local_{bits}} + local_{index} = indirect_{address}$. Thus, $f(x[2])$ will be routed to $y[6]$ with lane index 3 and local address 0 ($3 * 2^1 + 0 = 6$). Similarly, $f(x[3])$ will be routed to $y[3]$ which incidentally is the same lane as the origin (lane index 1) with local address 1 ($1 * 2^1 + 1 = 3$).

B. Performance Analysis

The quality of HLS tools is evaluated by measuring performance in terms of latency and initiation interval as well as FPGA implementation cost. The quality of an FPGA overlay NoC is evaluated by measuring the injection rate, sustained rate, worst-case routing latency along with FPGA implementation cost. How do we bridge the gap in terminology and meaning of these metrics across the HLS and NoC domains?

- Recall that initiation interval (II) is the rate at which loop iterations can be launched. This value is an integer greater than or equal to 1. It counts the number of cycles the inputs to the loop hardware must wait before the next loop iteration can be launched, and the inputs can be consumed. An II=1 is the fastest rate at which loop iterations can be processed. Loop-carried dependencies, or resource constraints may result in higher values of II than this ideal of 1.
- For NoCs, injection rate (IR) measures the intended rate of packet injection into the NoC and is achieved in absence of conflicts. It counts the number of cycles between consecutive packet constructions. The sustained rate (SR) metric measures the observed rate of packet injection into the NoC and is typically lower than IR due to packet conflicts. SR counts the number of cycles between consecutive packet injections into the NoC. Both IR and SR are numbers between 0 and 1 with 1 being the best result. In fact, this is another way of measuring initiation interval of the NoC, where $II = 1/SR$.

C. Cost Analysis

The cost of the hardware required to support indirect loops with crossbars or the NoC are a direct measure of the count and size of multiplexers needed in the design.

- When supporting loops with indirect addressing of the form shown in Figure 5 using existing HLS tools, the generated hardware includes wide multiplexers for each shared memory bank in the design. For a loop that must be unrolled by a factor N and arrays partitioned by factor N, this results in a crossbar-style organization of connections. Each copy of the loop datapath can connect to any memory bank. Thus, we need N multiplexers (one per bank) with N inputs each (one input from each datapath). Each multiplexer is routing the output of the datapath f with B bits. Thus, we need $N^2 \times B$ wires and $N \times\ N : 1$ multiplexers of width B.
- Hoplite NoC routers are dominated by the cost of multiplexers. Each router needs $2\times$ 2:1 multiplexers of size W where W is the width of the packet. For an $\sqrt{N} \times \sqrt{N}$ NoC, this results in a cost of $2 \times N$ 2:1 multiplexers of width W. When the NoC is used in lieu of the HLS-generated crossbar, $W = B$. It is important to note that the cost of the FPGA overlay NoC scales **linearly** with N instead of the quadratic N^2 cost that is necessary with conventional HLS.

D. Indirect Access Patterns

We partition the types of indirect accesses generated by HLS loops into multiple classes:

- **Destination Indirection**: This captures the class of accesses where the destination index is computed through an indirection *e.g.* $y[dest[i]] = f(x[i])$. Thus, the indirection is applied to the LHS (left hand side) of the dataflow expression. This pattern imposes fewer requirements of the NoC arbitration infrastructure and is implemented using the high-level organization shown previously in Figure 6. The values from the indirection $dest$ array become destination addresses for packets on the NoC. For this class of addressing, sequential consistency of parallel execution can be supported with additional resources[2] but generally not required for associative algorithms, nor expected by OpenMP weak-consistency [6] semantics.

- **Source Indirection**: This form of access inverts the indirection to the RHS (right hand side) of the expression *e.g.* $y[i] = x[src[i]]$ as shown in the code fragment of Figure 9. While this sounds like a simple change, the impact of NoC traffic is less obvious, particularly to a software developer with no hardware design experience, let alone NoC expertise. Source indirection generates Request-Reply style traffic pattern on the NoC that can lead to protocol deadlocks without virtual channel support. In a traditional NoC environment, two virtual channels are required to break this deadlock by routing requests and replies on two separate virtual channels. Hoplite does not support virtual channels to reduce FPGA implementation cost, and instead we choose

[2]An extra index RAM can be used to quash updates to y from packets reordered by the NoC. This will be investigated in future work.

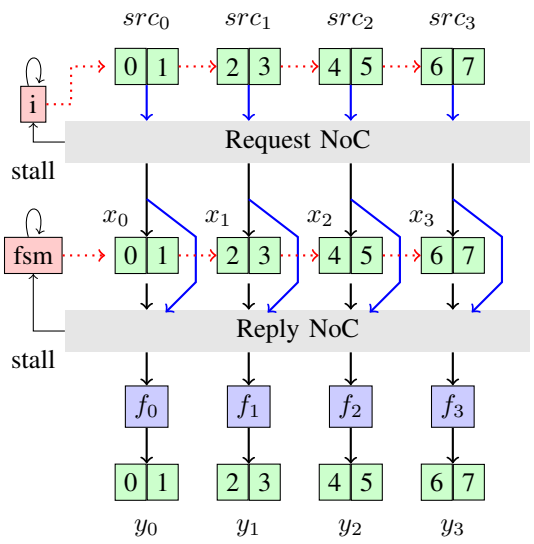


Fig. 8: Supporting source indirection using two physical NoCs for FPGA-friendly, deadlock-free implementation of Request-Reply NoC traffic.

to use multiple physical channels to support this class of traffic. It has been shown that support for virtual channels, associated buffers, and flow control logic is expensive on FPGAs by an order of magnitude or more [9]. Thus, a doubling of resources is a significantly cheaper alternative to increasing it by 10×.

```
for(int i=0; i<N; i++) {
  y[i] = f(x[src[i]]);
}
```

Fig. 9: Source indirection example.

We show a high-level organization of this design in Figure 8 for the code sketch in Figure 9. Here, the Request NoC routes NoC packets, from the src memory banks, carrying payload $src[i]$ to memory banks of x. Here, a lookup is performed to extract $x[src[i]]$ which is the input to the FPGA datapath f. This is routed back to the original bank for the index i over a physically distinct Reply NoC. The resulting packet is delivered and written into $y[i]$.

- **Nested Indirection**: In some cases, we may have nested indirection that requires a chain of lookups. For instance, Figure 10, shows a snippet of Sparse Matrix-Vector multiplication application with two levels of indirection through $offset$ and src. This is often the case for compressed sparse structures, pointer-based data structures, or sparse graph representations. This generates a dataflow ordering between packets on the NoC and also requires two deflection-routed physical channels for deadlock-free implementation.

IV. Evaluation

In this section, we first discuss our experimental setup and benchmarks used in our study. Next, we discuss the resource and performance tradeoffs for the LegUp-NoC designs across various benchmarks.

```
for(int i=0; i<N; i++) {
  y[i] = 0;
  for(int j=offset[i];j<offset[i+1];j++) {
    y[i] += a_nz[j] * b[src[j]];
  }
}
```

Fig. 10: Nested indirection example.

TABLE II: Benchmarks for different indirect access patterns.

Access Pattern	Label + Code Snippet	Applications
Destination Indirection	**indirect_dest** `y[dest[i]]=f(x[i]);`	Remote stores, Sorting, FFT, Scatter
Source Indirection	**indirect_src** `y[i]=f(x[src[i]]);`	Remote loads, Sparse Matrices, Gather
Nested Indirection	**indirect_nested** `y[i]=f(x[dest[src[i]]]);`	Sparse Matrices, Graph Analytics

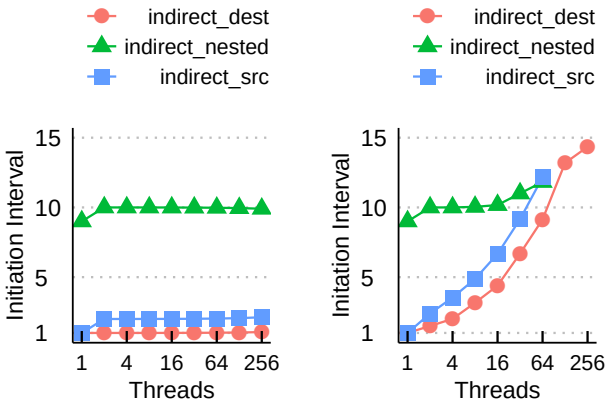
RTL: We use LegUp 5.0 for our work and build a LegUp compiler pass for processing loops with indirect accesses. We also evaluated Vivado HLS 2017.3 and observe similar trends as LegUp 5.0 due to the use of crossbar interconnect between datapath and memory banks along with static scheduling. Unlike Vivado HLS, LegUp 5.0 allows us to develop our own compiler pass that connects the FPGA datapaths to the memory banks with NoCs during the system generation phase of the compile flow. In LegUp 5.0, we use the Pthreads pass as it supports stalling the thread datapath in case on resource conflicts[3]. We develop a novel memory banking and partitioning pragma syntax to drive the interconnection process. The existing predicated banking support in LegUp 5.0 is not compatible with NoC-based design. Our approach is modular and allows simpler assembly of the NoC with the banks and datapaths. We use Modelsim 10.4d for cycle-accurate simulations to extract cycle counts for the different benchmarks. We compile the generated RTL with Vivado 2017.3 targeting the Virtex-7 485T (`xc7vx485tffg1761-2`) device from the VC707 board to extract post place-and-route metrics (area, frequency).

Benchmarks: We evaluate the effectiveness of the NoC-assisted HLS flow across various access patterns described in Section III. Each access pattern represents a class of application scenarios requiring indirect indexing. We evaluate these access patterns with a range of synthetic index distributions of the indirection arrays. We also consider workloads from real-world applications such as Sparse Matrix-Vector Multiplication (upto 16K nodes, 65K edges), Graph Analytics (upto 64K nodes, 131K edges), and 1-D FFT (1K–8K-point).

A. Initiation Interval (II) Trends

They key question we must answer is how the use of NoC affects the II of the benchmarks. The baseline LegUp II is always equal to threads (or unroll factor) due to static scheduling. First, we discuss results in the context of various access patterns with synthetic index traces.

[3]Not supported by the loop unrolling pass



(a) LOCAL indexing (b) RANDOM indexing

Fig. 11: Initiation Interval Trends for various access patterns and indexing distributions as a function of number of threads.

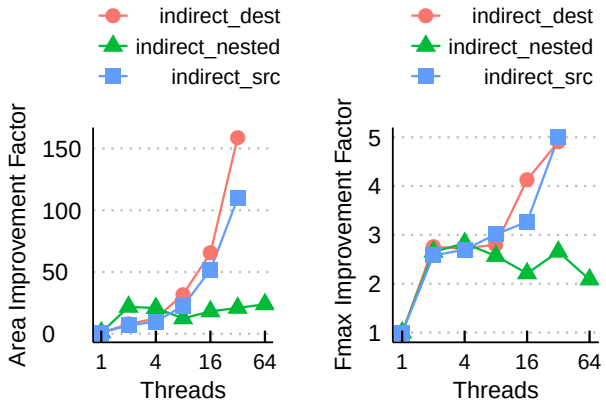
LOCAL indexing: In Figure 11a, we use a fully local indexing pattern where $dest[i] = i$. This trivial pattern is meant to confirm the baseline behavior of the datapaths.

- An II=1 is easily possible for simple destination indirection `indirect_dest` pattern as NoC traffic simply returns back to the sender directly.
- For source indirection `indirect_src`, we can only achieve an II=2. Recall, we need two separate physical NoCs to support Request-Reply NoC traffic pattern generated by the access pattern. It is not possible to accept back-to-back packets at the interface between Request and Reply NoCs as there is no guarantee of free slot in the Reply NoC in the subsequent cycle.
- For the `indirect_nested`, the II that we can achieve is 8–10. This is a limitation of LegUp 5.0 as it does not permit loop pipelining optimization when the loop indices are variable (not known at compile time).

RANDOM indexing: In Figure 11b, we quantify the impact of uniformly distributed indices $dest[i] = rand()$ on resulting II of the circuit. As we increase the thread count there is increasing contention in the FPGA overlay NoC resulting in higher II. At 256 threads, we see an II≈15 across all access patterns. Contrast this with LegUp behavior where II=256. This means that, with the NoC, we can launch loop iterations once every 15 cycles as compared to once every 256 cycles with baseline LegUp. Another important trend is how II of `indirect_nested` case catches up with rest of the access patterns at thread counts > 64. This suggests that the lack of pipelining does not hurt this access pattern at high thread counts. This also reveals an optimization opportunity for designs to save cost by generate smaller datapaths with a target II chosen to match the NoC's SR (sustained rate).

B. FPGA Resource and Mapping

The next question we must address is regarding the impact of the NoC on FPGA implementation costs and operating frequencies over baseline LegUp. In Figure 12, we quantify this effect on the physical implementation metrics of the FPGA (LUTs, Frequency). It should be no surprise that the NoC ($O(N)$) is more scalable than the Crossbar ($O(N^2)$). For `indirect_dest` and `indirect_src`, we see area gap as large as 150× at 32 threads. At 64 threads and above, we run out of FPGA capacity to implement the crossbar. For `indirect_nested`, the resource wins are only 25×, as the datapath dominates the NoC area. When considering Frequency, the large crossbars slow down the LegUp designs significantly by almost 5× at 32 threads. As before `indirect_nested` frequency is constrained by the datapath rather than the communication structures resulting in a lower, yet significant, 2–3× frequency win. When combined with the 16× improvement in II (RANDOM indexing), the overall speedups are as high as ≈50× over baseline LegUp. These numbers simple illustrate the overheads of pursuing a crossbar-based design strategy and we fully expect a smart FPGA developer would not intentionally choose this approach.



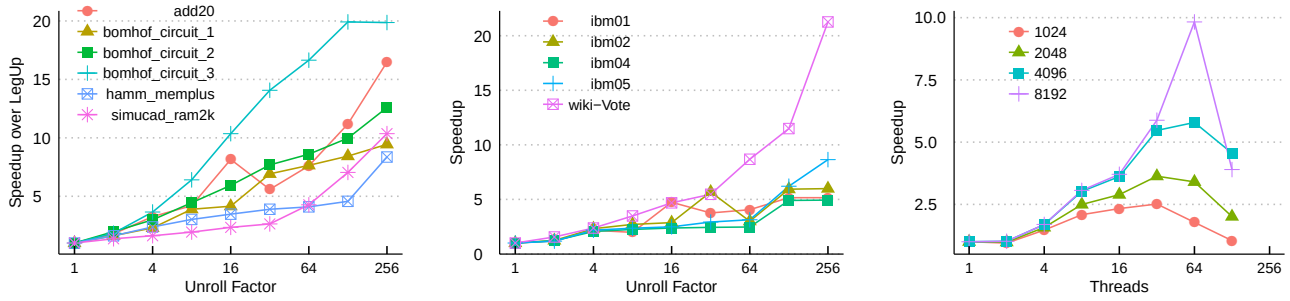
(a) FPGA LUT Cost Win (b) Frequency Win

Fig. 12: FPGA LUT cost, and Frequency trends for various access patterns when comparing NoCs against LegUp 5.0.

C. Real-World Benchmarks and Datasets

When considering real-world examples of indirect access patterns, how does our NoC-assisted approach hold up against LegUp? Real-world benchmarks include the indirect accesses as a component of their application, and we must understand how indirect accesses fit into the complete design:

- The SpMV and Graph benchmarks split computation across two phases. The first phase implements the indirect accesses to assemble data, and the second phase performs purely local operations in dataflow manner. In this case, the indirect accesses are routed over the NoC, while local operations do not need NoC function. The LegUp-generated 32b datapath for this implementation costs 278 LUTs + 343 FFs per thread which is larger than the NoC router (≈ 55 LUTs, 82 FFs) by ≈5× (Thus NoC is a 20% overhead over baseline area cost).
- An N-point FFT can be implemented as a sequence of $log(N)$ phases, with abundant parallelism within each level.



(a) SpMV (b) Graph Analytics (c) 1-D FFT

Fig. 13: Speedup trends for real-world benchmarks and datasets when using the NoC-enabled LegUp framework.

Data transfer between consecutive levels of the FFT is routed over the NoC using the butterfly pattern for that level. Thus, this benchmark reuses the NoC repeatedly in different application phases. For 32b fixed-point FFTs, the leaf-level butterfly building block occupies 186 LUTs and 172 FFs which is ≈3.3× larger than the NoC router (NoC is a 30% overhead over baseline area cost).

In Figure 13, we show the performance scaling trends across these benchmarks for various datasets (problem sizes) by measuring total application time (sum of cycles across all phases). We compute speedups against a sequential design to avoid penalizing the baseline needlessly with the avoidable overheads of a crossbar-based implementation. It is clear that the NoC helps deliver 5–20× speedups across the various designs. The quantum of speedup is a function of locality structure of the datasets. We observe diminishing returns over 64 threads primarily due to NoC bottlenecks. These speedups are different from the 8–16× speedup possible with the `RANDOM` indexing data seen earlier in Figure 11b due to effect of application locality and extra time spent in other non-NoC application phases. When considering only the NoC phase of operation of these benchmarks, the speedups are significantly higher at 15–35× suggesting the performance bottleneck is the LegUp-generated datapath pipelines.

D. Multiple NoC Channels

If the FPGA developer can afford to invest more resources in the communication infrastructure, there is an opportunity to use multiple NoC channels for improving performance. Multiple, concurrent NoC channels can provide parallel paths for the packets in the system to handle network congestion. In our design, we distribute packets across multiple channels in round-robin fashion, and retain the single packet injection/ejection constraint imposed by the client (datapath and RAM bank[4]). As shown in Figure 14, we see performance improvements of 2–3× (on top of existing wins) when using just two channels. This comes with a 20–30% increase in overall area as the NoC is smaller than the datapath blocks by 3.3–5×. The improvements are superior at larger system sizes due to lower contention at the NoC exit.

[4]Second port of the BlockRAM is reserved for HLS scheduling, and can unlock more improvements if exposed to the NoC

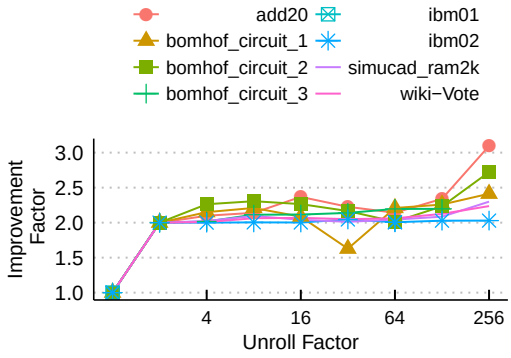


Fig. 14: Impact of two NoC channels on packet routing performance of the system. Improvement factors are larger at larger system sizes with lower NoC exit contention.

V. Discussion

A. Related Work

There have been various attempts at parallelizing loops with indirect addressing on FPGAs. We enumerate these ideas and highlight their limitations with respect to our work:

1. **Polyhedral techniques**: Loop tiling [2], [12] to determine bank configuration is possible for affine loops through suitable polyhedral analysis. This approach is only valid for loop with affine loop indexing (linear operations on loop index) that is not applicable to arbitrary indirection supported by our work where nothing is known about the access pattern `B[i]`. For large number of tiles, the memory banks may still need expensive wide multiplexers as per the results of the polyhedral analysis. Experimental results in these papers are limited to stencil-based workloads that only have nearest-neighbor communication pattern that is trivial to route on NoCs. Our technique applies to irregular graph-oriented workloads that are more difficult to analyze and parallelize.
2. **Trace-based analysis**: It is possible to mine the addresses [13] and analyze these address sequences to determine how to partition arrays and tile loops. This approach uses the address trace to prune the complexity of the memory bank crossbar to lower resource cost. With trace-based analysis, the presence of even a single indirection to

a bank requires the multiplexer to accommodate the link. In contrast, our approach uses a shared NoC and requires no static analysis of the addresses as they get resolved dynamically by the NoC through packet-switched operation. This paper uses datasets with regular communication patterns that would be easy to route over NoCs.

3. **Crossbar**: TILT-VLIW[11] employs statically-scheduled read and write crossbar across the various lanes of the VLIW architectures to enable parallel operation of the loop body across various ALUs. In contrast, our approach replies on an $O(N)$ scalable, cost-effective NoC and is dynamically routed requiring no knowledge of communication pattern. We have quantified the LUT and Fmax overheads of a crossbar-based solution in Figure 12.
4. **Configurable Connections**: EURECA [10] shows how to support dynamic indirect accesses through the use of configurable multiplexers, and dynamic generation of the multiplexer controls. This requires a modification to the underling FPGA architecture , while our approach works with existing FPGA chips through the insertion of overlay NoCs. We use packet-switching to determine NoC switch multiplexer controls while EURECA essentially time-multiplexes the hardware by supplying the per-cycle mux controls.
5. **Off-chip DRAM**: Our work complements other effort aimed at optimizing indirect accesses to single-ported external DRAM interfaces [5], [7]. While we focus on distributing indirect accesses across multi-bank on-chip RAMs, you can extend this idea to the design of memory controllers managing multiply address channels of external DRAMs and emerging multi-die HBM2-based architectures.
6. **FCUDA-NoC**: Prior work such as FCUDA-NoC [4] have supported interconnecting HLS-generated datapaths to a shared DRAM using expensive directory-based NoC routers. Their work does not address the opportunity for using HLS *within* the compilation itself, and focuses on system-level interfacing (DRAM) like others [1]. As shown in our paper, NoCs can become first-class citizens in the HLS toolflow.

B. Future Work

Going forward, NoCs in FPGA HLS can support routing of HLS AXI streams, OpenCL pipes, load-store shared-memory traffic, as well as operand routing between operators in dynamic HLS [8]. We can also support sequentially consistent operation by providing extra RAM and index checking hardware associated with each update. Further throughput improvements are also possible with the use of isolation buffers between the Request and Reply NoCs used in our design for source indirection.

VI. Conclusions

In this paper, we demonstrate how to integrate FPGA overlay networks-on-chip with high-level synthesis to provide scalable parallelization of loops with indirect addressing. NoCs provide a cheaper alternative than existing crossbar-based memory bank access structures, inferred by both LegUp and Vivado HLS, while delivering significant improvements to Frequency and Initiation Interval of the loops. Furthermore, we bundle this capability as a LegUp compiler pass thereby allowing the programmer to use NoC resources without NoC expertise. For various access patterns parallelized to 256 threads, we see as much as 16× improvement in II, 150× reduction in LUT cost, 4–5× increase in frequency for uniform random indexing. When applied to real-world benchmarks parallelized between 16–256 threads, we observe speedups of 5–20× for SpMV, Graph, and FFT benchmarks with a 20–30% overhead for introducing the NoC over baseline.

Acknowledgements: The authors would like to thank Jan Gray for providing access to Hoplite RTL source code, Jason Anderson and the LegUp team for providing access to and support for the LegUp 5.0 source code, and Stephen Neuendorffer for his comments about this manuscript.

References

[1] M. S. Abdelfattah and V. Betz. Networks-on-Chip for FPGAs: Hard, Soft or Mixed? *ACM Trans. Reconfigurable Technol. Syst.*, 7(3):20:1–20:22, Sept. 2014.

[2] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[4] Y. Chen, S. T. Gurumani, Y. Liang, G. Li, D. Guo, K. Rupnow, and D. Chen. FCUDA-NoC: A scalable and efficient network-on-chip implementation for the CUDA-to-FPGA flow. *IEEE Transact. on Very Large Scale Integration (VLSI) Sys.*, 24(6):2220–2233, June 2016.

[5] S. T. Fleming and D. B. Thomas. Using runahead execution to hide memory latency in high level synthesis. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 109–116, April 2017.

[6] J. Hoeflinger and B. de Supinski. The OpenMP memory model. *OpenMP Shared Memory Parallel Programming*, pages 167–177, 2008.

[7] L. Josipovic, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Trans. Embed. Comput. Syst.*, 16(5s):125:1–125:19, Sept. 2017.

[8] L. Josipovic, R. Ghosal, and P. Ienne. Dynamically-scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, New York, NY, USA, 2018. ACM.

[9] N. Kapre and J. Gray. Hoplite: A deflection-routed directional torus noc for fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 10(2):14:1–14:24, Mar. 2017.

[10] X. Niu, W. Luk, and Y. Wang. EURECA: On-chip configuration generation for effective dynamic data access. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 74–83, New York, NY, USA, 2015. ACM.

[11] K. Ovtcharov, I. Tili, and J. G. Steffan. Tilt: A multithreaded vliw soft processor family. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, Sept 2013.

[12] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[13] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang. A new approach to automatic memory banking using trace-based address mining. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 179–188, New York, NY, USA, 2017. ACM.