Study of Scatter/Gather & Inter-Neighbour Traffics Under FPGA Constraints

Siu Pak Mok (993870726) & Rafat Rashid (996096111)

December 23, 2012

Abstract

We evaluate the scatter-gather and inter-neighbour communication traffic for torus and tree topologies under FPGA constraints. To do this, we created a cycle accurate Simulator, Traffic Generator and implemented the Blocked LU Decomposition and Sudoku Constraint Propagation traffic algorithms. We have found the torus to be a better candidate for inter-neighbour traffic. Provided we connect the memCntrl intelligently to minimize the distance to it, scatter-gather traffic between compute nodes and memCntrl does not favour either topologies.

1 Introduction

In this report, we evaluate the scatter-gather and inter-neighbour communication patterns for the torus and tree topologies under FPGA constraints. The objective is to spend most of the time computing rather than on transferring data. To perform this evaluation, we have created a cycle accurate Simulator, Traffic Generator and implement two computationally intensive algorithms. We also propose future extensions to our work and things learnt.

1.1 Motivation

This project is motivated by Eric LaForest and Greg Steffan's work on Octavo which is a customizable ten-pipeline-stage, eight-threaded processor built on the Stratix IV FPGA [1]. A research direction currently being considered is the connection of multiple Octavos together to build a larger system. An interesting question then is how would data transfer between these Octavos.

Octavo currently does not have the infrastructure for us to build a multi-Octavo system and evaluate this question. In this report, we will not build this system but instead attempt to answer the following questions:

1. What kind of traffic patterns should we expect to see on Octavo and the FPGA?
2. What performance metrics should we consider for Octavo and the FPGA?
3. What are the constraints imposed on Octavo due to it being built on the FPGA?
4. What kind of topologies is best suited for the FPGA based on answers to questions 1-3?

We answer questions 1 to 4 in Section 2. Choosing a topology based on a given traffic is the subject of this report's simulation and evaluation component.

1.2 Our Contributions

We describe our contributions below. All components are implemented in C++.

Cycle Accurate Simulator Our Simulator executes custom traffic patterns on user parameterizable topologies. It accepts parameters to limit a node's Block RAM, I/O ports and the amount of data it can send or receive. As output, it records number of send, compute and idle cycles of each node as well as packets the node requests, sends and receives. These are recorded at every *stepSize* cycles to show the temporal variability of the traffic. Although we use the Simulator to study traffic under a simplified FPGA environment, we have built it to be easily extensible. New parameters, topologies and hardware constraints can be added without breaking existing functionality.

Simulated Applications We have used the Blocked LU Decomposition [2] for evaluating the scatter/gather traffic conditions and Sudoku Constraint Propagation [3] for analyzing inter-communication behaviour between nodes.

Graph Partitioning Algorithm We have implemented an algorithm to partition the work performed by our applications to the topology nodes.

2 Background

In 2003, Brebner and Levi proposed choosing a topology that best matched the application and the fixed programmable logic of the FPGA [4]. In 2006, Manuel Saldana evaluated five topologies (ring, star, mesh, hypercube, fully connected) with three network sizes (8, 16, 32) to examine how well they map to Xilinx FPGAs [5]. In

2007, the study was extended to the 2D torus and 64 nodes [6]. He focused on routing resources (wires), logic area utilization, maximizing operating clock frequency and the number of FPGA nets needed to place and route.

From his study, Saldana found that the underlying 2D mesh layout architecture of the FPGA does not benefit a particular topology. In fact, even fully connected networks is feasible when the number of nodes is kept small (below 16 nodes). The goal is to maximize the utilization of the existing FPGA resources. This is different from ASICs where the goal is to minimize the area and wires. Based on Figure 4 and 5 in [6], the logic (LUTs) and routing (nets) resource utilization exhibits a scalable linear trend for all topologies except for fully connected, which has a square trend. Most importantly, the utilization for the torus (although higher) is very close to that of the mesh and even the simplest ring network.

According to [6], fully connected topologies are unroutable beyond 22 nodes and the f_{max} degrades significantly compared to other topologies beyond 10. The ring topology meets the timing constraint of 150 MHz up to 63 nodes. Topologies beyond 63 nodes does not place and route due to insufficient resources. The torus and mesh meet timing up to 63 nodes and their f_{max} are extremely close (150.5 and 150.1), similar to the simple ring.

In 2010, Lee and Shannon have devised an accurate model that describes how topology parameters impact the performance of NoCs of up to 128 nodes (latency, bandwidth, operating frequency) on Xilinx Virtex 5 FPGAs [7]. They make a number of important observations: 1) topology choice has a greater impact on latency and bandwidth than resource usage, 2) resource usage does not impact performance below 80% when routability also fails, 3) link widths have a greater impact on performance than higher node degrees, 4) number of nodes has a bigger impact than node heterogeneity or their sizes [8] and 5) increasing node degree linearly reduces performance. Their next step is to bring this work to Altera FPGAs.

2.1 FPGA Constraints

From the above discussion, we draw the following conclusions. First, the choice of topology on the FPGA is limited by its performance and the application. For small networks, complex topologies like star, hypercube or fully connected can become favourable as they have better network latency and bandwidth characteristics than mesh or torus under certain conditions [7]. Moreover, we can take advantage of the FPGA's programmability to dynamically reconfigure the topology based on the particular communication patterns and needs of the application [9].

The BRAMs on a FPGA have only two ports for read and write [10]. If data routing shares the same memory with the processing elements, then the node will interfere with computation. This is because the processing element will need the ports for reading in operands for computation. FPGAs also have a fixed set of resources that exist whether they are used or not [5]. Therefore, ideally we should not limit the topology connectivity if there are resources available.

2.2 Related Works

Besides the works listed above, Bertozzi *et al* presents an ad hoc NoC synthesis tool called *NetChip* for MPSoCs that maps an application to a highly-parameterizable topology which can then be simulated [11]. Our work, although similar, is significantly more constrained.

3 Implementation

We describe the implementation of the Simulator and the Graph Partitioning algorithm below.

3.1 Simulator Design

Figure 1 shows a high level structural diagram of our cycle accurate simulator. The *Config* object parses a settings file to populate the *params* HashMap. Then an array of *SimpleNode* objects and the *MemoryCntrl* node is initialized. The method *SetupTopology()* initializes each node's routing table based on the provided topology parameters. Dijkstra [12] is used to find the shortest deterministic route. The table can be setup automatically for any topology once we manually define the directed graph (using each node's neighbours array). We are interested in the number of cycles needed to distribute the raw data, do the computation and write back to main memory.

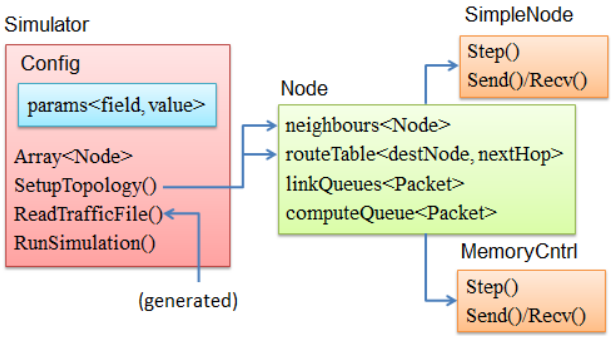

Figure 1: Simulator Structural Diagram

After reading in a Traffic file, the *RunSimulation()* method will be called. This method will iterate over each Node and call *Step()* until all nodes have nothing to do. *Step()* does one cycle worth of either computation or transferring data to neighbouring nodes using *Send()* and

Recv(). The *MemoryCntrl* scatters and gathers traffic to other "regular" *SimpleNodes*.

3.1.1 Step Function

The step function processes requests going into a node in the following order: incoming packets, outgoing packets and finally instructions. As the node does so, it uses up input and output resources defined in the config file. The config can define the number of reads (Read tokens) per cycle, write (Write tokens) per cycle and input/output ports (I/O ports) in the node. I/O ports act as buffers where data is queued, waiting to be saved into the BRAM. They are also a component of Octavo.

The node will attempt to retrieve data from the I/O ports first before reading from the BRAM. If it cannot, the node would then try to find the data in BRAM. If the data is found, a Read token is consumed. Similarly for writes, a Write token is consumed when the node does a computation or stores information on the I/O ports into BRAM.

Normally, each BRAM only has two ports to perform read and write operations. However, one can apply techniques such as multi-pumping, banking or other similar techniques to create RAM with more ports [10] [13].

3.1.2 Assumptions and Constraints

Asides from the FPGA constraints, we made several other assumptions to simplify our Simulator. First, BRAM, link and compute buffers are unlimited. This is for programming simplicity as we do not have to implement a way to evict cold cache lines. Also, we assumed the nodes to be smart. This means a node knows when the receiver has enough resources to handle the packet, thus eliminating the need for flow control.

Further, a node will respond to requests destined for another node if it has the requested data instead of propagating the request. This reduces the number of packets being sent and simulates the behaviour of what ideally should happen. All computations and wire delay take one cycle. The number of input/output BRAM ports is limited and can be changed via the config file. Finally, a node can send one packet to each of its neighbours per cycle.

3.2 Graph Partitioning

Any algorithm can be represented by a graph where the vertices represent computations and edges represent data dependencies. By partitioning the graph, one can distribute the workload across processors [14] and perform computations in parallel. Graph partitioning is also a popular technique used in the layout of digital circuits and components in VLSI [15]. We use this approach to generate traffic patterns for the simulation.

3.2.1 Algorithm Description

Graph partitioning is a well known NP-complete problem. There are no known efficient algorithms to partition the graph evenly while minimizing the number and weight of edges between partitions. Solving this problem is outside of the scope of this report. We use a simplified version of the Kernighan-Lin algorithm [16] instead.

We start with all the nodes in one large partition. Then we do a depth-first graph traversal and assign nodes along the path to a new partition. If the ancestors of a vertex is not assigned to the new partition, we assign them to the new partition. This process continues until the new partition has the same size as the one being partitioned. One can bipartition the graph further to create more partitions. Using this approach, vertices (computations) that depend on each other will likely be mapped to the same partition. After we have split the graph into the desired number of partitions, we add instructions to request for data if a parent of a node belongs to another partition.

4 Traffic Algorithms

The implementation of the Blocked LU Decomposition (LUD) and Sudoku Constraint Propagation (CP) is described below. We have verified their correctness using GSL's *gsl_linalg_LU_decomp()* function [17] and our Sudoku validation tool. Finally, we describe how the traffic file is generated from these algorithms.

4.1 Blocked LU Decomposition

LUD is the decomposition of a square matrix into a (L)ower and (U)pper triangular matrix. We use the blocked variant of the algorithm to simulate the scatter/gather traffic pattern. The algorithm is computed inplace, resulting in a matrix that consists of the L and U with the diagonal values of the L matrix normalized to 1.

As described in Wei Zhang's paper [2], there are three common versions of the blocked algorithm. We have implemented the "right-looking" version. The matrix is divided into blocks of which there are four types (Figure 2a). As shown in Figure 2b, the block operated on (black) depends on the top-most and left-most blocks (grey).

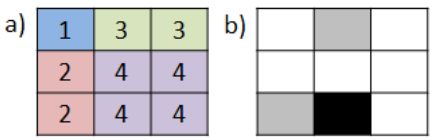

Figure 2: LU Block Types

The computation proceeds as shown in Figure 3. Note that as soon as the Type 1 block is computed, all the Type

2 and 3 blocks can be computed in parallel. Type 4 blocks can be computed as soon their dependent left-most and top-most blocks are computed.

We observe scatter/gather characteristics inherent in moving the dependent and compute blocks to and from off-chip memory and the cores to perform the computation. It would make sense for each block to be computed by a separate core to take advantage of the caching behaviour. Preferably, the block size should be small enough for three such blocks to fit into a core's cache.

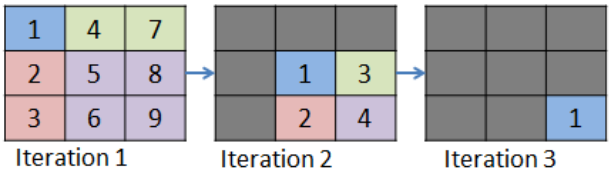

Figure 3: LU Blocked Algorithm

4.2 Constraint Propagation

Sudoku is a number-placement puzzle game. The standard puzzle is a table made up of 9 rows, 9 columns and 9, 3x3 boxes. The puzzle starts with given numbers in various positions and the player's goal is to complete the table such that each row, column and box contains every number from 1 to 9 exactly once. CP is the first step used in solving a Sudoku. It consists of two rules:

Rule 1: For any cell, if a number already exists in its row, column or box (the cell's peers), the possibility of that number for that cell is removed.

Rule 2: For any cell, if all of its peers has a specific number removed, the cell itself must contain that number.

By repeatedly applying these two rules, the possible values a cell can take is gradually minimized and more singletons are discovered. CP does not guarantee that the puzzle will be solved. More elaborate methods are used to augment CP to solve difficult puzzles. These rules are described by Peter Norvig [3]. For the puzzles we use, CP is sufficient to solve them.

4.3 Traffic File Generation

The Simulator requires a traffic file that we generate using our traffic and graph partitioning algorithms. This is accomplished in two steps. First, we generate a Compute file by adding output statements for all computations in the traffic algorithm. We use a HashMap to map memory addresses of variables to their version number. The first time we encounter a variable, its version number is initialized to 1. Every time this variable is written to, its version number is incremented. Figure 4 illustrates this for a traffic that finds the sum of three variables.

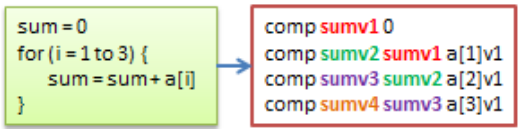

Figure 4: Compute file generation from C++ code

Mem Cntrl	Node 1	Node 2
send node1 a[1]v1	(wait)	(wait)
send node2 a[2]v1		
send node1 sumv1		
requ node3 sumv3	comp sumv2 sumv1 a[1]v1	requ node1 sumv2
(wait)	send node2 sumv2	
	(wait)	comp sumv3 sumv2 a[2]v1
		send memCntrl sumv3
(DONE!)		(wait)

Figure 5: Traffic generation from Compute file

By doing this, we can visualize a data flow graph which is used by our graph partitioning algorithm to distribute the traffic between topology nodes and perform computations in parallel. Figure 5 illustrates this partitioning of work for finding the sum of two variables. The 'requ' command is used to request data from a node. The concatenation of the instructions shown in Figure 5 results in the traffic file that is read by our Simulator.

We initially planned to use LLVM [18], a research-oriented compiler, to compile programs into a Data Flow Graph (DFG) and then apply a partitioning algorithm to generate a traffic flow. There are several benefits for doing this. First, it would allow us to simulate programs that are complex in nature. Also, it provides an automatic way to generate traffic patterns for any topologies.

Ilian Tili, a masters student under Professor Greg Steffan has written a tool that generates a DFG from a C++ program using the LLVM compiler. We have successfully used his tool for simple programs. However, it has some limitations. The tool can only handle floating point operations and does not support complex array indexing or C++ loop constructs. As a result, we instrument the C++ file to generate the Compute file instead.

5 Evaluation

In Section 2, we noted that given there is sufficient resources on the FPGA, the choice of topology should be based on the expected traffic. Here we present the results for a 4x4 torus and 16 node tree and analyze their effectiveness against our two proposed traffic patterns.

5.1 Experimental Setup

For all graphs, the x-axis represents number of cycles normalized by *stepSize*. At every *stepSize* cycles, we record

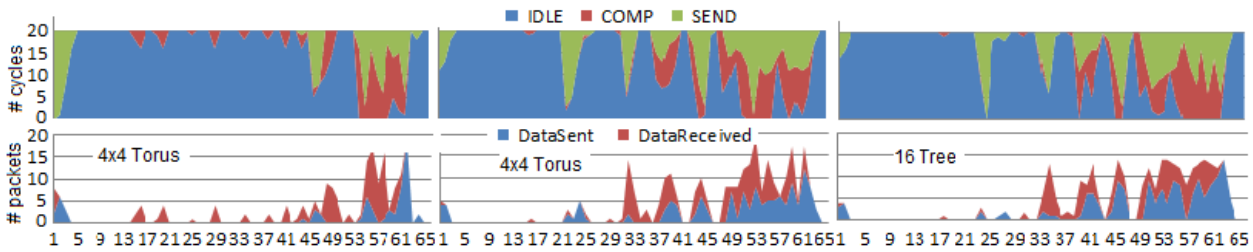

Figure 6: Node 2 - Left: 15x15 Matrix, blkSize 5, Middle: 16x16, blkSize 2, Right: 16x16, blkSize 2; *stepSize* 20

what occurred during that period. The *memCntrl* is connected as shown in Figure 7. For all nodes, *InsnsPerCycle* is set to 1, *IOPorts* to 4, *BRAMWritePerCycle* to 1 and *BRAMReadPerCycle* to 2 to respect FPGA constraints. For the *memCntrl* node, *PacketsSendPerCycle* and *PacketsReceivePerCycle* are set to 4.

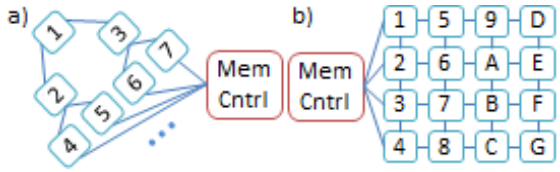

Figure 7: Simulated Topologies

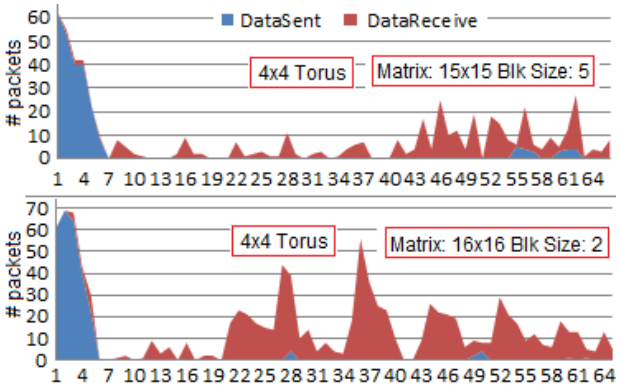

Figure 8: LUD: MemCntrl - *stepSize* 20 cycles

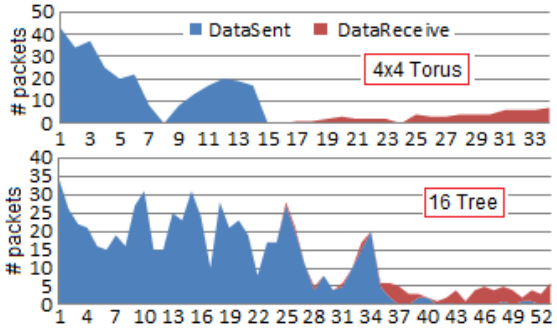

Figure 9: CP 16x16 Sudoku: MemCntrl - *stepSize* 20

5.2 Evaluation Results

For LUD, lots of data is initially sent from the *memCntrl* to populate compute nodes (Figure 8). After a block is computed, it is returned to the *memCntrl*. Recall intermediate nodes can send data on behalf of the *memCntrl* if they have a copy of it (Section 3.1.2). This significantly reduces the *memCntrl*'s need to keep sending latest data. Also, with a smaller block size, sending/receiving is much more frequent. Computed data is sent to the *memCntrl* earlier. For the torus, the 15x15 and 16x16 (with a smaller block) matrices took 1323 and 1317 cycles respectively. We observed a similar behaviour for the tree topology, taking 1309 and 1345 cycles.

The left and middle graphs in Figure 6 compare using a block size of 5x5 and 2x2 on the torus. Generally, a node spends more time being idle when a larger block is used. For the 15x15 matrix, this makes sense as 16 compute nodes have only 9 blocks to work with in the first LUD iteration and most are dependent on others being computed first. The network traffic of moving around blocks to and from the *memCntrl* is also more significant. With a smaller block, more can be computed in parallel. However, the amount of required computation also increases. This pattern is also true for the tree topology (15x15 graph for tree is omitted for space constraints).

In LUD, blocks are returned to the *memCntrl* every time they are computed. Differently, in CP (Figure 9), only the unknown cells are sent back when they are solved. This explains the much lower *DataReceive* events. However, to compute these cells, the entire puzzle must be sent to the compute nodes. If the *memCntrl* is attached to only the tree's root, a very large spike in send events is observed initially. This dissipates sharply after the first 7-10 *stepSizes* as closer nodes start to forward the puzzle. This is also observed for the torus. Since in Figure 9, the *memCntrl* is attached to four nodes in the middle of the tree, more packets are sent from the *memCntrl* as it is often closest.

Figure 10 shows representative results for Node 4 in the torus. Figure 11 does the same for the tree. Unlike LUD, compute nodes spend a lot of time sending/receiving data

between them, as expected. This is because every time CP reduces the possible values that any cell can take, this information has to be propagated to other compute nodes before they can proceed to the next step in CP.

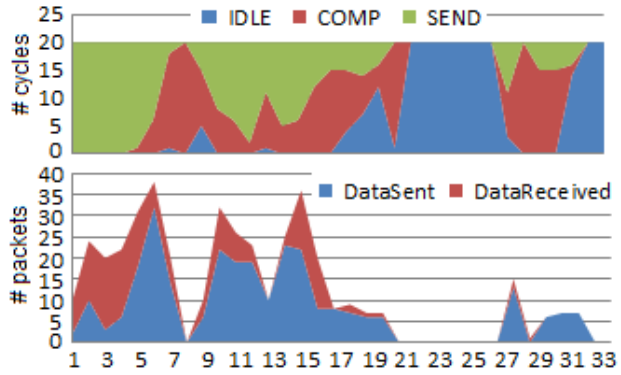

Figure 10: 16x16 Sudoku, 4x4 Torus: Node4, *stepSize* 20

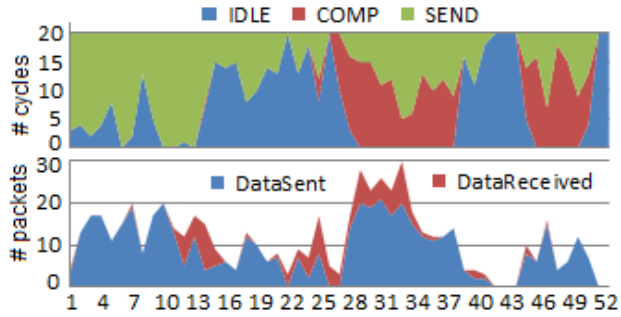

Figure 11: 16x16 Sudoku, 16 Tree: Node4, *stepSize* 20

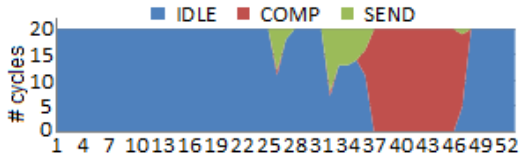

Figure 12: 16x16 Sudoku, 16 Tree: Node16, *stepSize* 20

Node 4 is connected to the *memCntrl* directly in both topologies. Interestingly, all nodes in the torus exhibit similar compute and traffic behaviour to Node 4. This can be attributed to the regularity of the torus. This is not true for the tree where the behaviour is much more disproportionate. The tree's root node spends much fewer cycles sending data and more time being idle. This behaviour is similar for Node 16 (Figure 12). Like the root, it is furthest away from the *memCntrl* and has fewest ways to get to the node. This is reflected in the much larger number of cycles that it takes to simulate the 16x16 Sudoku for the tree, 54 *stepSizes*, compared to 35 for the torus.

The behaviour described is similar for 9x9 Sudoku. It takes 40 *stepSizes* on the torus and 49 on the tree. Surprisingly, for LUD, *stepSizes* taken for torus and tree differ by at most two for both 15x15 and 16x16 matrices. We believe this is because in LUD, once a node receives a block, it can compute it independently of other nodes. Conversely in CP, inter-neighbour communication is frequent and can bottleneck due to the frequency of data that needs to be sent or received from nodes like the root and Node 16 that are farthest away and have no path diversity.

From these observations, the torus seems to be a much better candidate for traffic patterns that exhibit a lot of inter-neighbour traffic. Conversely, for algorithms that have communication between only the *memCntrl* and a compute node, both torus and tree can be used so long as the *memCntrl* is connected intelligently to the tree.

6 Future Work

We require a better method for partitioning traffic. Our current approach does not do a good job of spreading out the workload evenly. It is also not conscious of spatial and temporal locality of data. Since we already instrument a C++ program, we could add more directives to generate a more realistic traffic. For example, forcing the LU algorithm to compute a block on only one core. However, this approach is not very scalable as the user would need to add annotations to a program and understand the data movement behaviour in order to use the Simulator.

The BRAM, link and compute buffers should also be finite and parameterizable via the config file. A cache eviction protocol should be implemented to increase the accuracy of the simulation. This will allow us to investigate the impact of varying the nodes' buffer sizes on performance. Further, routing policies, including adaptive variants should be considered for future work.

We want to add support for variable link latency and prioritize nodes closer to the *memCntrl* when scheduling computations to reduce latency and network communication. Finally, we would like to extend Ilian's work to automate traffic generation and eliminate C++ annotations.

We have successfully simulated a 128x128 matrix with 32 block size on the 4x4 torus and 16 tree. The results has been omitted from this report to simplify the analysis. After applying the improvements proposed above, we would like to scale up our evaluation to larger and more diverse sets of topologies and traffic patterns. We want to maximize the resource usage of a modern FPGA.

7 Conclusion

We have created a cycle accurate Simulator, Traffic Generator and implemented two traffic algorithms to evalu-

ate the scatter-gather and inter-neighbour communication patterns for the torus and tree topologies under FPGA constraints. We have found the torus to be a much better candidate for inter-neighbour traffic. Provided we connect the *memCntrl* intelligently (such as multiple center nodes instead of only to the root of the tree), scatter-gather communication between compute nodes and the *memCntrl* does not favour either topologies.

Acknowledgements

Thanks to Ilian Tili for help with setting up LLVM and using his tool. His source, DFG.cpp, can be found within the source package attached with this report.

References

[1] C. LaForest and J. Steffan, "Octavo: an fpga-centric processor family," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 219–228.

[2] W. Zhang, V. Betz, and J. Rose, "Portable and scalable fpga-based acceleration of a direct linear system solver," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 1, p. 6, 2012.

[3] P. Norvig, "Solving every sudoku puzzle." [Online]. Available: http://www.norvig.com/sudoku.html

[4] G. Brebner and D. Levi, "Networking on chip with platform fpgas," in *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*. IEEE, 2003, pp. 13–20.

[5] M. Saldaña, L. Shannon, and P. Chow, "The routability of multiprocessor network topologies in fpgas," in *Proceedings of the 2006 international workshop on System-level interconnect prediction*. ACM, 2006, pp. 49–56.

[6] M. Saldana, L. Shannon, J. Yue, S. Bian, J. Craig, and P. Chow, "Routability prediction of network topologies in fpgas," *IEEE Transactions on VLSI Systems: Special Section on System Level Interconnect Prediction*, vol. 15, pp. 948–951, 2007.

[7] J. Lee and L. Shannon, "Predicting the performance of application-specific nocs implemented on fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 23–32.

[8] L. Shannon and J. Lee, "The effect of node size, heterogeneity, and network size on fpga based nocs," in *International Conference on Field-Programmable Technology*. IEEE, 2009, pp. 479–482.

[9] T. Bartic, J. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, "Topology adaptive network-on-chip design and implementation," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 152, no. 4. IET, 2005, pp. 467–472.

[10] C. LaForest, M. Liu, E. Rapati, and J. Steffan, "Multi-ported memories for fpgas via xor," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 209–218.

[11] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "Noc synthesis flow for customized domain specific multiprocessor systems-on-chip," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 2, pp. 113–129, 2005.

[12] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[13] C. LaForest and J. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 41–50.

[14] B. Chamberlain *et al.*, "Graph partitioning algorithms for distributing workloads of parallel computations," *University of Washington Technical Report UW-CSE-98-10*, vol. 3, 1998.

[15] S. Ravikumār, *Parallel methods for VLSI layout design*. Greenwood Publishing Group, 1995.

[16] S. Kernighan, B. W. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal*, vol. 49, pp. 291–307, 1970.

[17] "Lu-decomposition reference manual," *GNU Scientific Library*. [Online]. Available: www.gnu.org/s/gsl/manual/html_node/LU-Decomposition.html

[18] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.