Routing Packets for Multi-FPGA Communication: A Case Study Accelerating Krylov Series Computations over GF(2)

***Abstract*—Technological advances have enabled FPGA-FPGA communication to assume reliable multi-gigabit data transmission rates. As the consensus grows in the design community and IP vendors towards a standard network-on-chip (NoC) interface, we attempt to explore the benefits of an additional abstraction layer, based on the established framework of NoC, to achieve packetized communication between multiple FPGAs. In particular, we harness existing NoC generator to automatically partition the network across multiple FPGAs and recast the existing design flows of multi-FPGA prototyping and multi-FPGA accelerators based on our automated multi-FPGA NoC partitioning. This is further elucidated by our hardware implementation of boolean matrix vector multiplication, whose scalability is assisted by our ability to partition its underlying network interface over several FPGAs.**

I. Introduction

The demands on more logic capacity, on-chip dedicated resources, I/O and high-speed transceivers etc., has been rising even as the FPGA application domains grow more pervasive; and are being met in large part due to Moore scaling. More than Moore scaling too has long been happening at the system level integration via multi-FPGA platforms. Emulation and verification of large ASICs, and Hardware acceleration, are the two broad application categories where multi-FPGA platforms have been traditionally used. Today, full system rapid or pre-production prototyping also frequently employs multi-FPGA platforms [1] [2]. The design flows involving multi-FPGA systems, often custom designed, present with serious challenges not just in technology integration but a particularly tough one at the design automation level. The multi-FPGA partitioning problem is still considered to be one of the toughest challenges in FPGA prototyping/emulation, and is further exasperated with advanced ASICs consisting of over a billion gates, and particularly time-consuming clustering and partitioning approaches. In practice, the process is cumbersome and error-prone, requiring manual intervention and knowledge of the partitioned modules.

In addressing the challenges noted in this context, we believe that a design flow mapped over a *standardized* and *simplified* communication framework, such as the pervasive packet-switched Network-on-Chip (NoC) framework, together with a automated partitioning step of such a network for multiple FPGAs, could greatly allay the designers of the tedious setup phase of multi-FPGA design automation. The partitioned communication network should also be flexible in terms of the choice of topology and router configurations.

With our automated partitioning flow, as a proof-of-concept study, we present the case of hardware acceleration of boolean matrix vector multiplication occurring in the context of Krylov sequence computations, where the scalability of the communicatin intensive algorithm is determined by the ability to partition its network over several FPGAs. Although in incipient stage, our approach helps raising the level of abstraction in multi-FPGA design scenario, which has remained an esoteric art of a few hardware designers despite its numerous advantages.

The remainder of the paper is organized as follows. In the next section, we outline two design flows which could benefit by our approach. We then present our solution to the multi-FPGA partitioning problem and a discussion of the implementation details and advantages in Section III. In Section IV we report a case study of an important algorithm in the context of high-performance computing, along with evaluation results and a discussion. Section VI discusses some related work, and Section VII concludes.

II. Pertinent Design Flows

Our proposed multi-FPGA partitioning scheme over a packet switched communication framework is likely to benefit the following two design flows.

A. Hardware acceleration

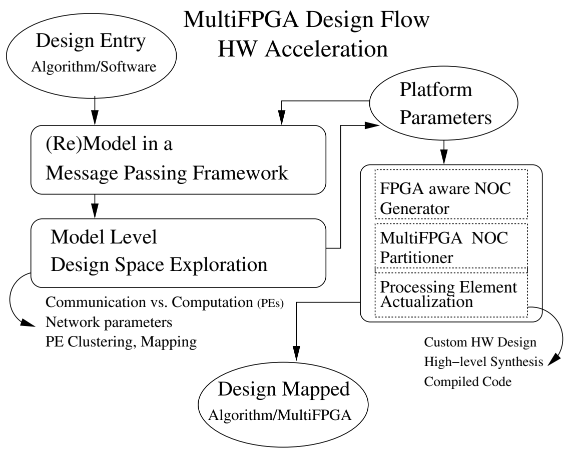


Fig. 1: Design flow applicable for Hardware Acceleration

Figure 1 outlines a design flow for multi-FPGA hardware acceleration. The design entry is usually an application/algorithm that can benefit from hardware acceleration, scaling over multiple FPGAs. The next phase is to express/re-model the algorithm in a framework of software threads—corresponding to processing elements in hardware—communicating in a message passing fashion. The target platform parameters such as the topology of choice, number of FPGAs could be selected to suit the model of the algorithm, with an awareness of constraints such as link-bandwidths, and resources.

It is to be noted that the effort here is comparable to that of mapping to commodity accelerator architectures such as CUDA [3]; only here, the broad parameters of the accelerator architecture, including scaling, are customizable by the designer. A software implementation of the model, for instance, could be done with MPI [4]. The next optional step is a design space exploration at this model level to profile for the computation and communication traffic behaviors. The information at this stage could be used for processing element clustering (mapping to an FPGA) and partitioning (across FPGAs); the wealth of research on PE mapping (e.g. [5]) reported in the context of NoCs could be leveraged here.

The next phase of the automation is hardware generation and has two steps: Processing element actualizations, and Network generation and partition. The network with the chosen parameters (topology, router configuration, etc.,) is generated by an FPGA architecture aware NoC generator, which is followed by partitioning of the NoC across the available number of FPGAs. Actualizing the processing elements (PEs)—corresponding to the software threads of the message passing model—could be done via custom designed soft IPs or can be obtained through a High-level synthesis flow such as Vivado[6]. A PE could even be a custom processor executing the corresponding cross-compiled code.

B. ASIC prototyping

At least at the modeling level (e.g. [7]), there have been efforts (from both IP vendors and integrators) towards standardized IP/module interface in the context of Network-on-Chip based SoC integration. If [8] the component modules (processors, CODECS, MODEMS, etc.) and subsystems (memory, IO) of the system use standard interfaces for communication over a packed-switched NoC, the multi-FPGA [rapid] prototyping of MPSoCs that integrate such components could be achieved using the proposed design automation. Even those MPSoC designs not so large as to require multiple FPGAs could still be effortlessly partitioned across a multi-FPGA platform: this enables faster design to debug cycles owing to localization of bugs and partial reconfigation of specific FPGAs.

Also, unlike hardware acceleration which could potentially use arbitrarily large number of FPGAs, many ASIC [rapid] prototyping cases need less. And in this context, we note that the proposed scheme is particularly suitable for single-package multi-FPGA solutions such as Virtex 7 series FPGAs enabled by Stacked Silicon Integration [9].

III. Multi-FPGA Partitioning of Custom NoC

A. Problem

While extending the NoC methodology for achieving communication between multiple FPGAs seems to be a compelling idea, there are multiple challenges which need to be addressed in this regard. We list down some of the important challenges here:

1) *First*, NoCs frequently use large port-width (consisting of few 10s of bits) for communication between interconnected routers, but the number of I/O pins on an FPGA is severely limited to meet this requirement. This concern is further aggravated in high radix topologies, where the NoC routers have high fan-out.
2) *Second*, the inter-FPGA interconnect delays are often high and also highly variable, especially when the different FPGAs are residing on separate boards. This makes it harder to achieve reliable communication using NoC.
3) *Third*, it is hard to have a common clock reference across multiple FPGAs without the use of Phase Locked Loops (PLLs) and synchronizers. This further complicates the design. Moreover, modules on different FPGAs could be optimized towards different operating speeds, in which case it may be suitable to operate them on different clock domains. This adds further complications to communicate in a synchronous manner.

In order to resolve these challenges, we use asynchronous serial communication between the partitioned networks. This not only minimizes the requirement of a large number of parallel I/O pins for communication between partitioned networks, but also eliminates the need for a common clock reference. To this effect, our partitioning problem can be stated as: "Given a set of interconnected routers and input/output ports, partition the network into disjoint subsets of routers, with each partition containing the specified set of routers, inter-connected in the same fashion with routers and the associated I/O ports within the partition, and insert and expose appropriate serial interface for interconnections across the partition." We require that the additional interfacing logic at network partitions be such that the application modules using the network interface and the network routers themselves must remain oblivious to the partitioning. Figure 2 provides an an example of NoC partitioning of four routers on two FPGAs, with communication between the FPGAs using a serializer/deserializer interface.

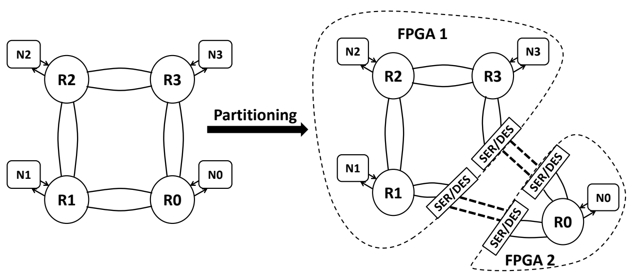


Fig. 2: Example partitioning of an NoC with four routers on two FPGAs. The router R0 (along with its processing element N0) is mapped onto a separate FPGA. Communication between FPGAs takes place using serializer/deserializer (SER/DES) interface.

B. Implementation Details

Dally et. al [8] recently advocated for standard NoC generators for modularized designs to improve the design productivity. Separation of application development from the communication infrastructure having a standard interface improves the development time as well as the testing and verification costs. We use a freely available web-based synthesizable RTL generator for custom Network-on-Chip (NoC), named CONNECT(Configurable Network Creation Tool). CONNECT [10] can be used for generating a NoC of arbitrary topology and supports a large variety of router and network configurations. Also, CONNECT incorporates a number of useful features fine-tuned for the FPGA platform, as opposed to the other popular NoC generators that are primarily intended for ASIC development.

As stated previously, we require asynchronous serial interface for across-the-FPGA communication. For this purpose, we use a variant of Universal Asynchronous Receiver/Transmitter (UART) protocol in which 16 bits (instead of 8 bits) are serially transmitted and received in a single frame. The UART transmitters and receivers contain shift registers operating nominally at the same frequency (typically much lower than the operating frequency of the hardware), known as the baud frequency, generated by a baud generator. This is also the sampling frequency for the receiver. In general, the data clocks at the transmitter and receiver side could be different and the sampling clock could be provided by the local baud generators, as long as the baud rate remains the same.

Our script automatically partitions an input NoC into separate sub-networks which can be inter-connected via this serial interface.

Since flow control signals are also required to cross the FPGA partition along with the serial data links, we would like to minimize the number of flow control signals required in view of the limited I/O pin availability on a single FPGA. For this reason, we use peek flow control over credit based flow control. Peek flow control requires only a single bit signal per port for flow control. Additionally, peek flow control eliminates the need for maintaining registers for storing credits. We also do not use Virtual Channels (VCs) although it helps maximize network performance. This is also to minimize the FPGA pin count requirement, since flow control occurs on a per-VC

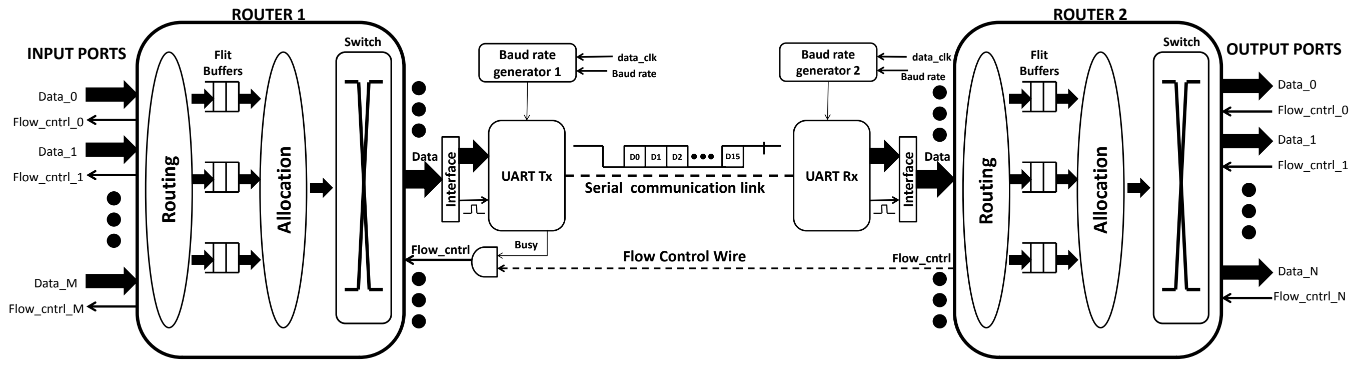


Fig. 3: Communication between routers using UART. Dotted lines indicate the wires crossing the interface.

basis. Also note that our approach can be easily integrated within the CONNECT NoC generator to directly generate partitioned sub-networks, with the input parameters provided in advance.

We now describe the additional custom logic required to interface asynchronous serial communication using our UART modules with the CONNECT generated NoC in a manner that the rest of the modules remain unaffected. First, individual flits in CONNECT carry information about the destination address. This information also needs to be communicated to receiving routers over serial links along with the data bits. So, the data transfer port on the UART transmitter is connected to the lines corresponding to the destination bits and lowest significant bits of the outgoing flt data, such that the total size equals 16 bits (for sizes larger than this, the frame size must be further increased). Similarly, on the receiver side, the bits carrying the destination address to the receiving router are appropriately connected to the corresponding bits on the received data port of UART receiver. On the transmitter side, the representing a valid flit signal is connected to the input of the UART corresponding to the valid data signal to prompt the UART transmitter to start transmitting the incoming data. The upstream router does not assert its valid flit signal when the UART transmitter is busy or when the downstream router is not ready (through the incoming flow control signal indicating full buffer). Similarly, the UART receiver prompts the downstream router to receive incoming flit when the receiver has completely received the serially transmitted data frame. Figure 3 shows an example of two routers along with the additional UART circuitry. In practice, the SER/DES links[11] are to be implemented over the many (Multi Giga Bit Transiever) MGT links; this is a work in progress.

C. Advantages

The following is an enumeration of some advantages of the proposed approach.

- The approach enforces *modularity* and *information hiding* in a multi-FPGA design, further improving design productivity and development time of multi-FPGA designs. The user-modules are oblivious to the NoC partitioning infrastructure, and the application developer is not required to be aware of details of serialization/deserialization details of the implementation.
- Scales seemlessly with the number of partitions and modules.
- The partitioning is flexible in terms of choice in topology and routing owing to CONNECT FPGA optimized NoC generator.
- This NoC-based partitioning approach allows several modules operating at different clock speeds to effortlessly communicate with each other.In fact, our approach could be used in prototyping voltage-frequency islands for *GALS*-based networks-on-chip onto FPGAs as described in [12]. The authors have demonstrated 40% energy savings using this approach on real video applications.

IV. Case Study: Boolean Matrix Vector Multiplication

Boolean matrix vector multiplication (BMVM) has important applications in coding theory and cryptanalysis , among others. Several secure systems today rely on the computational intractability of factoring large numbers [13] and the Number Field Sieve (NFS) (Lenstra, [14]) is perhaps the fastest known factorization algorithm. Of the four major steps in NFS, *Sieving* and *Matrix Step* are considered to be the most time-consuming steps in the NFS [15]. In particular, the Matrix Step involves finding linear dependencies in a large boolean matrix. Block Wiedemann [16] algorithm is often used for this purpose. This translates to several Krylov sequence computations $(Av_i, A^2v_i, ..., A^rv_i)$ involving a very large boolean matrix A (dimensions in the range $10^6 \rightarrow 10^{11}$), and a large number of random boolean vectors v_i, with i ranging from 1 to r, where $r = 2D/K$. Here, D is the column size of matrix A and K is known as the "blocking factor".

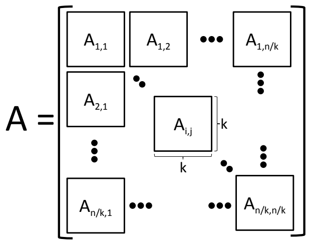
Our choice of BMVM as a case study has been influenced by a couple of factors, other than our research interest in accelerating iterative BMVM for NFS. First, our BMVM method is an instance of communication-intensive workload which benefits from the NoC design methodology. It's scalability is also ameliorated by our ability to partition network over multiple FPGAs. Secondly, we have encapsulated our hardware implementation in a software library using a standard CPU-FPGA integration tool which demonstrates how software development could benefit through multi-FPGA acceleration.

A. Previous work

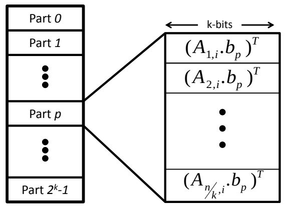
Geiselmann et. al [17] proposed a hardware implementation of the "mesh routing" algorithm for matrix step in NFS using several interconnected ASICs. Bajracharya et. al [18] proposed an implementation of the above mesh routing circuit on a reconfigurable architecture. In mesh routing, a matrix A of size $n \times n$, requires a mesh of size $m \times m$ nodes, where $m = \sqrt{n.h}$, h being the maximum number of non-zero bits in any column of matrix A. The $m \times m$ nodes are further divided into D blocks, each of size $h \times h$, which are initialized with the input matrix v and would produce the product vector v' at the end of the algorithm. The h nodes in a block i are required to store the addresses (requiring $log(n)$ bits) of the non-zero bits in column i. For computing the product, each non-zero block would "route" a message to target blocks corresponding to the h addresses of that block, with each receiving block i, flipping its single-bit product value (initialized to 0 before routing starts) corresponding to i-th bit in v' on reception a message. [18] have reported calculations indicating 1024-bit matrix step factorization possible in 27 days using 1024 FPGAs. Our method is better suited for handling "dense" matrices and requires only sub-quadratic number of operations w.r.t. input matrix dimension by virtue of matrix pre-processing.

B. Method: Sub-quadratic algorithm to BMVM

Our approach is based on the recently proposed combinatorial algorithm for matrix vector multiplication by Ryan Willams [19]. This approach involves a one-time pre-processing step on A, enabling a sub-quadratic time computation of BMVM.



(a) Tiling matrix A



(b) Composition of the LUT_i

Fig. 4: One-time pre-processing phase

The one-time pre-processing phase involves partitioning the matrix A into tiles of dimensions $k \times k$ as in Figure 4a, followed by construction of n/k look-up tables $\{LUT_i \mid i : 1 \rightarrow n/k\}$ corresponding to each of the n/k columns of the tiled A. LUT_i stores all possible linear combinations of columns of each $k \times k$ tile in the column i of the tiled matrix A (Figure 4a). There can be 2^k linear combinations of columns of each $k \times k$ tile, and there are n/k such tiles in a column of A. Figure 4b shows the composition of LUT_i, which is partitioned into 2^k parts, each part storing n/k $k-$bit words such that part$-p$ stores vectors $\{A_{1,i}b_p, A_{2,i}b_p, ..., A_{n/k,i}b_p\}$, where b_p is the k-bit vector corresponding to the partition index p. In short, the pre-processing step is equivalent to pre-computing and storing all possible products of the tiles of matrix A (ie., $A_{1,1}, A_{1,2}..A_{n/k,n/k}$) with any k-bit vector.

The computing phase uses this pre-processed information to compute Av, for some vector v. Let v be likewise partitioned into n/k *sub-vectors* $(v_1^T, v_2^T, .., v_{n/k}^T)$, and let $v' = Av = (v_1'^T, v_2'^T, .., v_{n/k}'^T)$. For illustration, let LUT_i, and v_i^T be with processing node-i (or thread-i). As $v_i' = A_{i,1}v_1 \oplus A_{i,2}v_2 \oplus \ldots \oplus A_{i,n/k}v_n/k$, if each processing node-i looks-up partition indexed by v_i in LUT_i, and send each of the n/k words stored in this partition to the corresponding processing nodes, the result v_i' at each processing node-i is obtained by XOR-accumulating all the incoming k-bit messages.

In [19], the authors have used $k = \epsilon log_2(n)$, with $\epsilon < 0.5$, so that the multiplication can be carried out in $O(n^2/(\epsilon log(n))^2)$ on a $log(n)$-word RAM. The implications of choice of parameter k are summarized below.

Word width of LUT	k
Number of LUTs	n/k
Size of each LUT (in words)	$2^k.(n/k)$
Number of word operations required	n^2/k^2

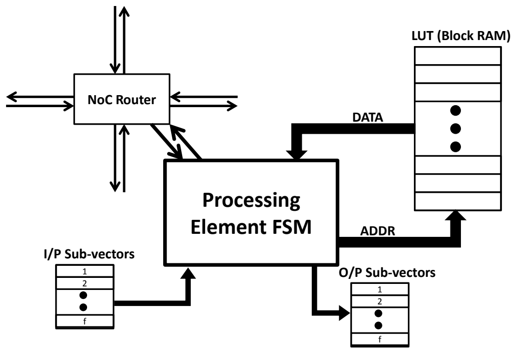


Fig. 5: Processing element.

C. Implementation Details

The pre-processing stage is one-time and has been done as described above, in software. The three components involved in this mapping are: the processing node architecture, the network infrastructure and the FPGAs. As our processing nodes need to contain LUTs, which are efficiently mapped to the onchip Block RAMs available in abundance on modern FPGAs (Virtex 6, for instance, has large number of $36Kb$ BRAMs totalling upto $38Mb$). Depending on the problem parameters (n and k), not all processing nodes can be mapped to a single FPGA. As per our earlier discussion, we map all the n/k processing elements across all the FPGAs in our NoC-driven multi-FPGA platform. It is important to ensure that while multiple such messages may simultaneously attempt to update a particular product sub-vector v_i', the updates are appropriately serialized to maintain correctness. Since only one flit can be injected and ejected in a single cycle in the NoC, this constraint is automatically ensured. Our implementation supports the following "Network and Router Options" for NoC generated using CONNECT (topology and number of endpoints is user-specified):

Router Type	Simple Input Queued (IQ)
Flow Control Type	Peek Flow Control
Flit Data Width	16
Flit Buffer Depth	8
Allocator	Separable Input first Round-Robin

Since number of sub-vectors can be very large (n/k), we also implement "folding", such that a single processing element is handles multiple sub-vectors and is provided with a single coalesced look-up table corresponding to the input sub-vectors. The extent of this folding is referred to as the "folding factor" f. Figure 5 shows a single processing element, and figure 6 shows the top module consisting of several such processing elements connected by a CONNECT generated NoC, which can also be partitioned over multiple FPGAs. Our final step is to integrate this setup with software to make it usable as a FPGA-accelerated software library. We achieve this using RIFFA 2.0 [20]. Figure 7 shows the overall framework. Our implementation also supports iterative BMVM to compute $Av, A^2v, ..., A^rv$, with number of iterations r communicated by the software endpoint to the FPGA along with the input vector v.

V. EXPERIMENTAL RESULTS

For experimental evaluation, we implemented our hardware on Xilinx Virtex 6 ML605 XC6VLX240T-1FFG1156 Evaluation Board using Xilinx 14.3 ISE, with software endpoint running on a 1.6

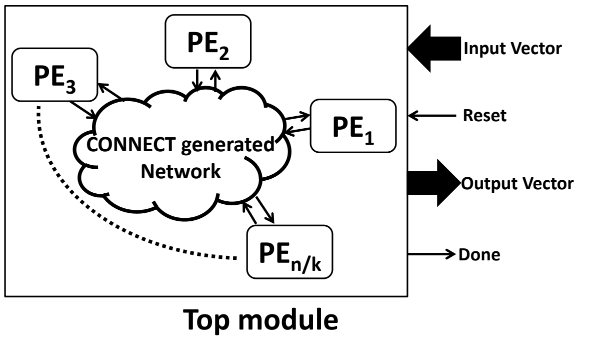


Fig. 6: Top module for Boolean Matrix Vector Multiplication (BMVM).

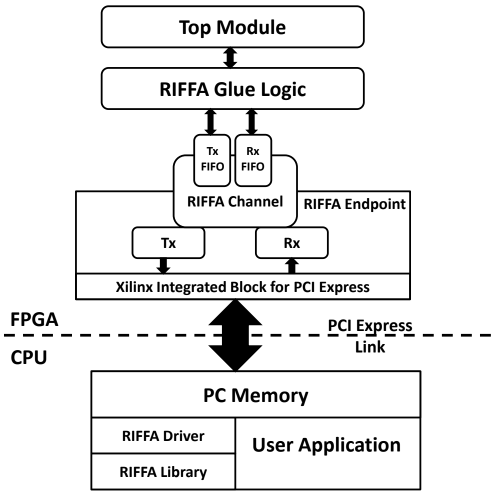


Fig. 7: BMVM framework with RIFFAa.

GHz Intel i7-2600 processor. We have also implemented a multi-threaded equivalent software implementation of the described algorithm using the *pthread* library, with each thread corresponding to one precessing element in its hardware equivalent. Concurrent updates of multiple threads are serialized using *locks* and *barriers* are used in between successive iterations. Our multi-threaded software BMVM was evaluated on a 1.2 GHz six core (dual thread) Intel Xeon E5-2620 processor (Model 45, 15360 KB cache) and was used as a baseline for comparison. Our hardware modules were made to operate at 100MHz clock, with partitioned networks using a UART interface with baud rate of 115200 bits/second. We evaluate our implementation on following two configurations:
Configuration 1: $n = 64, k = 8, f = 2$

Configuration 2: $n = 1024, k = 4, f = 4$

Configuration 1 requires four processing elements (threads) for a input matrix of size 64×64. As a proof of concept, we partitioned the input 2×2 mesh on two separate FPGAs. We have reported the execution time (including roundtrip over RIFFA interface) for partitioned as well as unpartitioned mesh. As can be observed from the result tables, the partitioned mesh results in a slowdown due to the slow speed UART bottleneck. We are working on using faster interconnection links to get speedup results comparable to single cycle unpartitioned mesh.

Configuration 2 requires 64 processing elements for a matrix of size 1024×1024. We have evaluated the results for four network topologies implemented on a single FPGA with single cycle hop between adjacent routers, which depict a clear correlation between network cost and performance (the cost increases moving from ring to mesh to torus to fat tree but performance also improves accordingly). When number of iterations are low (1-10), the overheads in terms of host processor - FPGA communication time in hardware and thread *creation/join* time in software, are a dominant component of the overall execution time. For larger iterations (100-1000), the actual computation times dominate and the total execution time increases nearly linearly with number of multiplication iterations.

VI. Related work

Network-on-Chip has been a topic of immense interest in the last decade. A rich body of work exists on several themes concerning NoC, such as application-specific topology, routing, micro-architecture, power, placement, verification, among others. Multi-FPGA partitioning for prototyping of large designs dates back to the pre-NoC era, when the FPGA was a lot more resource constrained. Ouaiss et. al [21] developed ”SPARCS”, a synthesis and partitioning tool for multi-FPGA systems, with shared memory or direct channel communication between partitioned tasks. Roy-Neogi et. al [22] suggested a genetic algorithm based partitioning method for multiple FPGAs within resource and timing constraints. Papamichael et. al [10] developed an automated NoC generator tool tailor-made for FPGAs. Liu et. al [23] and Mostefaoui et. al [24]have developed a multi-FPGA emulation framework, with MGT transceivers for communication. However, the system only “mimics” NoC, and does not implement it on FPGA.

Our work bears most similarity with Fleming et. al [25], who were perhaps the first to propose a packetized communication mechanism for multiple FPGAs having automated network partitioning. However, their work is more narrowly focused around automated partitioning for a class of latency-insensitive hardware designs that conform to a specific model of computation. They also do not use serial communication between FPGAs, which is why their pin count requirement would be large. Stepniewska et. al [26] have also worked on a similar idea using high speed serial I/O on FPGA.

VII. Conclusion and Future work

Multi-FPGA partitioning has historically been considered a tough challenge, owing to the difficulty in contriving communication interfaces between separate FPGAs. Our automated approach to Network-on-Chip partitioning, based on a standard, freely available NoC generator, not only placates this challenge, but also extends the pervasive NoC design methodology to the multi-FPGA scenario. Our Boolean Matrix Vector Multiplication (BMVM) application demonstrates the utility of our partitioning technique for scalability and prototyping of large hardware designs. Our FPGA implementation of BMVM can achieve orders of speedup compared to its software counterpart, even at modest clock frequencies, if provisioned with high-speed interconnects. Our preliminary investigations make a case for NoC-based FPGA designs for hardware acceleration and prototyping. This work opens up avenues for research and lays a strong foundation for further exploration of some pertinent ideas. Our ongoing work aims to extend our methodology to use faster interconnect technologies for inter-FPGA communication and calibrating our boolean matrix vector multiplication algorithm to handle large and sparse matrices.

Iterations	Time (in msec)			Speedup (Norm. to Software)	
	Software	Mesh	Partitioned Mesh	Mesh	Partitioned Mesh
1	0.32	0.052	1.48	6.15	0.22
10	1.1	0.052	14.14	21.15	0.078
100	5.2	0.087	140.3	59.8	0.037
1000	44.2	0.58	1402.8	76.2	0.031

TABLE I: Comparative results for configuration 1 (average over 100 experiments).

Iterations	Time (in msec)					Speedup (Norm. to Software)			
	Software	Ring	Mesh	Torus	Fat_tree	Ring	Mesh	Torus	Fat tree
1	4.0	0.205	0.075	0.060	0.052	19.5	53.3	66.7	76.9
10	22.9	1.67	0.412	0.299	0.275	13.71	55.58	76.6	83.3
100	204.3	16.15	3.64	2.83	2.33	12.7	56.12	72.2	87.7
1000	2025.4	160.51	35.60	28.09	22.69	12.6	56.9	72.1	89.3

TABLE II: Comparative results for configuration 2 (average over 100 experiments).

References

[1] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel nehalem processor core made fpga synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, (New York, NY, USA), pp. 3–12, ACM, 2010.

[2] J. Ray and J. C. Hoe, "High-level modeling and fpga prototyping of microprocessors," in *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pp. 100–107, ACM, 2003.

[3] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[4] "MPI: A Message-Passing Interface Standard Version 3.0," 09 2012.

[5] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular noc architectures," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, pp. 551–562, April 2005.

[6] "Xilinx, inc. vivado hls," 2013.

[7] "Open core protocol." www.ocpip.com.

[8] W. J. Dally, C. Malachowsky, and S. W. Keckler, "21st century digital design tools," in *Proceedings of the 50th Annual Design Automation Conference*, p. 94, ACM, 2013.

[9] "Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency," tech. rep., Whitepaper WP380, Xilinx Inc., 12 2012.

[10] M. K. Papamichael and J. C. Hoe, "Connect: Re-examining conventional wisdom for designing nocs in the context of fpgas," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 37–46, ACM, 2012.

[11] A. Athavale and C. Christensen, "High-speed serial i/o made simple," *Xilinx Inc*, vol. 4, 2005.

[12] U. Y. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu, "Voltage-frequency island partitioning for gals-based networks-on-chip," in *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pp. 110–115, IEEE, 2007.

[13] M. J. Wiener, "Cryptanalysis of short rsa secret exponents," *Information Theory, IEEE Transactions on*, vol. 36, no. 3, pp. 553–558, 1990.

[14] A. K. Lenstra, H. W. Lenstra Jr, M. S. Manasse, and J. M. Pollard, "The number field sieve," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pp. 564–572, ACM, 1990.

[15] C. Anand and S. II, "Factoring of large numbers using number field sieve-the matrix step," 2007.

[16] D. Coppersmith, "Solving homogeneous linear equations over (2) via block wiedemann algorithm," *Mathematics of Computation*, vol. 62, no. 205, pp. 333–350, 1994.

[17] W. Geiselmann and R. Steinwandt, "Hardware to solve sparse systems of linear equations over gf (2)," in *Cryptographic Hardware and Embedded Systems-CHES 2003*, pp. 51–61, Springer, 2003.

[18] S. Bajracharya, D. Misra, K. Gaj, and T. El-Ghazawi, "Reconfigurable hardware implementation of mesh routing in number field sieve factorization," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pp. 263–270, IEEE, 2004.

[19] R. Williams, "Matrix-vector multiplication in sub-quadratic time:(some preprocessing required)," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 995–1001, Society for Industrial and Applied Mathematics, 2007.

[20] M. Jacobsen and R. Kastner, "Riffa 2.0: A reusable integration framework for fpga accelerators," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pp. 1–8, IEEE, 2013.

[21] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures," in *Parallel and Distributed Processing*, pp. 31–36, Springer, 1998.

[22] K. Roy-Neogi and C. Sechen, "Multiple fpga partitioning with performance optimization," in *Field-Programmable Gate Arrays, 1995. FPGA'95. Proceedings of the Third International ACM Symposium on*, pp. 146–152, IEEE, 1995.

[23] Y. Liu, P. Liu, Y. Jiang, M. Yang, K. Wu, W. Wang, and Q. Yao, "Building a multi-fpga-based emulation framework to support networks-on-chip design and verification," *International Journal of Electronics*, vol. 97, no. 10, pp. 1241–1262, 2010.

[24] Kouadri-Mostefaoui, Abdellah-Medjadji, B. Senouci, and F. Petrot, "Large scale on-chip networks : An accurate multi-fpga emulation platform," in *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pp. 3–9, Sept 2008.

[25] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer, "Leveraging latency-insensitivity to ease multiple fpga design," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, (New York, NY, USA), pp. 175–184, ACM, 2012.

[26] M. Stepniewska, A. Luczak, and J. Siast, "Network-on-multi-chip (nomc) for multi-fpga multimedia systems," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pp. 475–481, IEEE, 2010.