Multi-FPGA Hardware Acceleration of Boolean Matrix Vector Multiplication using Network-on-Chip

DUAL DEGREE DISSERTATION

Submitted in partial fulfilment of
requirements for the degrees of

Bachelor of Technology
(Electrical Engineering)

Master of Technology
(Microelectronics)

by
Yatish Turakhia
Roll no. 09D07015

Under the supervision of
Prof. Sachin Patkar

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
DEPARTMENT OF ELECTRICAL ENGINEERING

JUNE 2014

Approval

This dissertation entitled **Multi-FPGA Hardware Acceleration of Boolean Matrix Vector Multiplication using Network-on-Chip** by **Yatish Turakhia** is approved for the degrees of **Bachelor of Technology** and **Master of Technology**.

Examiners

_____

_____

_____

_____

Supervisor

_____

_____

Chairman

_____

_____

Date: June 25, 2014
Place: Indian Institute of Technology Bombay, Mumbai

Declaration

I declare that this written submission represents my ideas in my own words and wherever others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/ data/ fact/ source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Indian Institute of Technology Bombay and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Yatish Turakhia
Roll No. 09D07015

Date: June 25, 2014

Acknowledgments

I would like to express my deepest gratitude towards Prof. Sachin Patkar, Electrical Engineering Department, for his persistent guidance and invaluable inputs throughout the course of this project.

I would also like to thank Vinay Kumar for his insightful ideas during this work. I am also thankful to Mr Kiran R, M.Tech student at EE department, IIT Bombay, and Mr. Kartik Lakhotia, for their generous help with some software tools. My heartfelt thanks to my batchmate Mr. Shardul Gaur, and all other members of the HPC group, who have provided immense cooperation throughout this project. This work would not have been possible without the support of all of the above.

Contents

List of Figures

List of Tables

Abstract

A strong trend towards hybrid CPU-FPGA architectures has been impelled by the rapid increase in transistor density and high customizability of FPGAs, which have been traditionally viewed as prototyping platforms for ASICs. There is an urgent need to extend the highly pervasive Network-on-Chip (NoC) methodolgy for development of FPGA designs. In particular, a NoC-like interface for seamless multi-FPGA communication is highly desired. This work automates the NoC-partitioning for multiple FPGA using a standard NoC generator tool. We also built a flexible and modular FPGA-accelerated library foundation for iterative Boolean Matrix Vector Multiplication (BMVM) kernel using the NoC framework. The scalability of our implementation is assisted by our ability to partition a NoC over several FPGAs. Our preliminary results indicate about 100X speedup of BMVM using our hardware over software implementation, even with modest clock frequencies.

Chapter 1

Introduction

1.1 Thesis Objectives

The following are the objectives of this work:

Automated Network-on-Chip partitioning for multiple FPGAs
Rising integration costs, owing to technology scaling, has enabled designers to integrate tens to hundreds of IP blocks on a single chip. A shared bus interconnection severely limits the performance scaling in this scenario, since it serializes the parallel bus accesses for communication. On-chip packed-switch interconnection of micro-networks, known as Network-on-Chip, is being seen as a viable solution to the involuted problem of on-chip communication. On the other hand, the FPGAs, which have a more lasting scaling roadmap, are gaining strength for performing power-efficient hardware acceleration, in addition to the prototyping purposes for ASICs. Our goal is to extend the NoC design methodology to the FPGAs and automate its partitioning for seamless communication between the FPGAs.

FPGA-based hardware acceleration of Boolean Matrix Vector Multiplication Boolean Matrix Vector Multiplication is a vitally important kernel in several scientific and numerical computation algorithms. Our goal is to accelerate iterative Boolean Matrix Vector Multiplication using FPGA, in which a pre-processed matrix is used several times over for different vector multiplications, and interface it with software to make it usable as a flexible off-the-shelf library function.

1.2 Organization of Thesis

Following provides the outline of the subsequent chapters in the report:

Chapter 2 provides a precis to the theory of Network-on-Chip (NoC) and the details and unique features of the NoC generator tool used in our work.

Chapter 3 defines the multi-FPGA NoC partitioning problem and provides the details to our approach in automating the partitioning process.

Chapter 4 expounds the theory of boolean matrix vector multiplication (BMVM) and a recently proposed sub-quadratic algorithm to perform BMVM, with some pre-processing. This chapter also explains our implementation details of the FPGA hardware, inspired by the algorithm, and the way it is integrated with software to perform FPGA-accelerated iterative BMVM. A multi-threaded software implementation of the same algorithm is also discussed, which is used for comparison.

Chapter 5 describes our experimental setup and methodology used to obtain the comparative results of our implementation.

Chapter 6 provides a brief summary to the related work in the past literature in the context of multi-FPGA partitioning and boolean matrix vector multiplication.

Chapter 7 finally concludes the report with some discussion on the possible future research directions in the context of this work.

Chapter 2

Network-on-Chip Overview

This chapter would provide the readers a basic understanding of the Network-on-Chip (NoC) design methodology. For those conversant with this topic may skip to Section 2.2 to understand the fundamentals of CONNECT, the NoC generator tool used in our work. Readers may also refer to [10, 24] for further understanding of NoC and to [23] for further details on CONNECT tool.

2.1 Introduction

Many core chip multi-processors are increasingly being seen as the new incarnation of high performance computing. Next generation of multi-core processors may contain hundreds of processors on a single chip. As the number of cores keeps rising, the on-chip communication infrastructure needs to revamp from the traditional bus architecture, which is the major performance bottleneck. Network-on-chip (NoC) is increasingly being seen as a possible alternative as it achieves the necessary trade-off between the cost-efficient but performance-limited bus architectures and performance efficient but cost/area inefficient point-toâĂŞpoint architectures. It can achieve a high degree of parallelism, however, since its design space is large, the designer is required to make intelligent design decisions to meet the Quality of service (QoS) requirements.

At a high level, a Network-on-chip (NoC) design methodology consists of a number of computational resources communicating to each other over a network infrastructure. The computational resources may include general purpose processing cores, DSP cores, specialized H/W modules, memory units such as RAMs, or even a heterogeneous combination of the above. The communication infrastructure consists of a set of interconnected routers (sometimes also referred to as "switches") which relay the messages

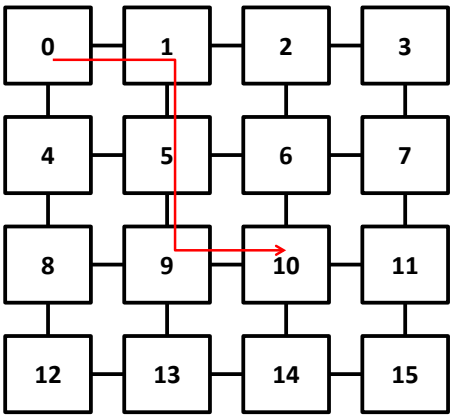


Figure 2.1: An example routing on a NoC.

from the sending computational element (also referred to as processing element (PE)), in the form of smaller "packets" and "flits", to the destination computational element. In the NoC nomenclature, a "packet" is the smallest unit used for routing and sequencing and may be composed of one or more "flits". A flit (flow control digit) is the unit of bandwidth and storage, and is often also the smallest unit that can be transferred over a channel from one router to the other in a single cycle. It must be noted that a flit may be required to traverse over a number of network routers (which are also responsible for the routing decisions) before reaching the final destination. In effect, the routing of messages, in the form of smaller packets and flits, takes place on a network infrastructure, consisting of routers and interconnects between them, while the computational elements (connected to the network infrastructure through their associated routers) are only responsible for performing the required computations based on these messages. Figure 2.1 shows an example of a set of 16 routers connected in a *Mesh* topology, in which a flit sent by router 0 traverses through three other routers (1, 5 and 9) before reaching its final destination, namely router 10.

The NoC methodology is largely inspired by the established “layered" approach in computer networks and effectively involves separation of application development from the communication infrastructure, thereby ameliorating design productivity. The pioneering paper of Dally et. al [10] originally proposed the use of an on-chip network instead of global wiring for achieving communication between modules. Kumar et. al [18] further dwelled on this idea and identified the *four* layers in the NoC design methodology:

1. The **physical layer** : This layer specifies the width of the router buffers and channels.

It also determines the *phit-width*. This layer is *technology-dependent*.

2. The **data link layer**: This layer determines the protocol (for example single cycle or pipelined data transfer) between two interconnected routers. This is again *technology-dependent*. This layer may also incorporate error-checking.

3. The **network layer**: This layer specifies the routing and forwarding mechanism of packets from the sender to receiver node. The processing nodes are indexed by the address bits (encoded within packets). This is again *technology-dependent*.

4. The **transport layer**: This layer is responsible for packetization of messages into smaller "packets" and "flits" and also the appropriate addressing of source and destination nodes. This is often handled by the interfacing modules. This layer is *technology-independent*.

As we enter the *billion-gate* era [9], the NoC design methodlogy significantly helps to improve the productivity, design cost and time-to-market in the following ways:

- The dissociation of communication and application layers and restricting the communication to standard interface enforces *modularity* in design, which also aids reuse of modules. It enables communication between heterogeneous modules and also imposes *information hiding*, since details of one module need not be known to the other for communicating messages. This has been explained in [9].

- The isolation of the network interface further reduces the *test* and *verification* costs.

- The NoC can be easily customized for a particular application to meet the *timing* and *performance* requirements.

- This is a *scalable* approach and can be easily extended to hundreds of communicating processing elements.

In addition to the above, NoCs also find applications in *cache coherency* [5], *GALS* architectures [21] and *secure* architectures [11]. We now discuss the four major considerations governing the design of a Network-on-Chip, as identified in [24], namely the NoC *topology*, *routing*, *flow control* and *router microarchitecture*:

1. **Topology**: The *topology* of a network describes how the routers and the processing elements are connected in the network infrastructure. Most commonly explored NoC topologies in the literature include Ring, Mesh, Torus, Tree, Butterfly and fully-connected. Figure 2.2 shows some of these commonly used topologies. The white

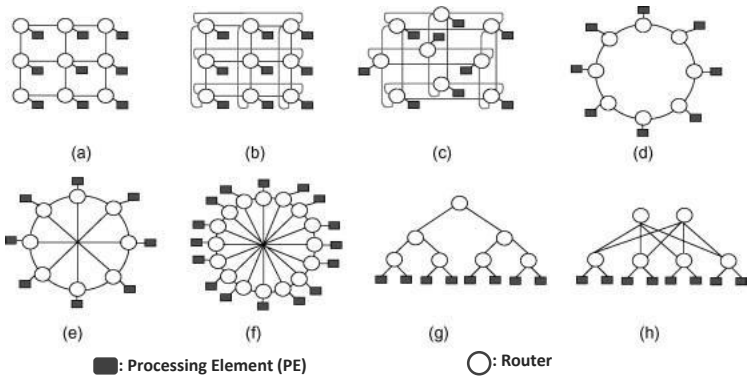


Figure 2.2: Common NoC topologies: (a) Mesh (b) Torus (c) Folded Torus (d) Ring (e) Octagon (f) Spider (g) Binary tree (h) Butterfly Fat Tree.

circles represent routers and black squares represent processing elements (PEs). Topology can have a significant impact on the overall performance, as topology, together with routing strategy, determines the number of hops for message transmission. A topology often determines the cost of the NoC as it decides the number of PEs used and layout complexities involved. It may also have a significant bearing on the NoC power consumption, which is increasingly becoming a serious concern.

2. **Routing**: The *routing* strategy determines how the packet is routed from its source to the final destination. The routing algorithm maybe (i) *deterministic*, in which the path from source to destination is always fixed (e.g. X-Y routing in Mesh), (ii) *oblivious*, in which routing decisions maybe different for the same source and destination, however, the routing decision does not consider the state of the network (e.g minimal oblivious routing) or (iii) *adaptive*, which uses local or global information about the network state to make decisions (e.g. PQ-routing, negative first routing). The goal of the routing strategy is often to balance the load in a network, however, several routing strategies suffer from deadlock and the designer needs to be careful in choosing the right routing strategy.

3. **Flow Control**: The role of *flow control* is to regulate and manage the shared resources, such as buffer space, in the network. Since, unlike computer networks, the NoC is required to be lossless, the flow control protocol must also ensure that packets are not dropped and reliably delivered. Based on the granularity of resource allocation and bandwidth utilization, the flow control mechanisms are broadly classified as *store-and forward*, *virtual-cut through*, *wormhole* and *virtual channel* flow control

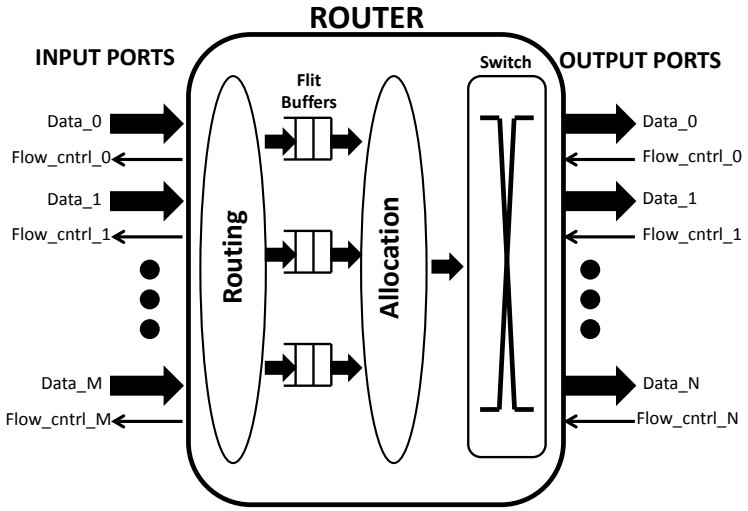


Figure 2.3: A typical router micro-architecture.

(more details available in [24]). Flow control mechanism has a direct impact on the packet latency (and therefore Quality-of-Service or QoS), link utilization and complexity of router micro-architecture. The most commonly used flow control mechanisms for guaranteeing lossless behaviour of network are (i) *credit-based* flow control and (ii) *peek* flow control (also referred to as on/off signalling). In credit-based flow control, the sending routers maintain a credit count for each downstream router, which indicates the amount of available buffer space in the downstream routers. In peek flow control, the downstream routers directly signal the buffer availability and the upstream routers are required to check this signal before forwarding flits. Peek flow control obviates the need for maintaining credit counters for available buffer space and also reduces the number of flow control signals.

4. **Router micro-architecture**: The *router micro-architecture* deals with the actual circuit implementation of the router that helps in realizing the routing and flow control strategies. A typical router consists of a set of input ports connected to a routing logic, which determines the output port along which the flit/packet has to be forwarded. The flits are stored in buffers before being scheduled by the arbitration logic to the switching network, after considering the buffer space availability in the downstream buffers. The switch finally helps the packet depart the router through the selected output port. Additionally, each port (input and output) has flow control signals associated to it. Figure 2.3 show a typical router micro-architecture. Some-

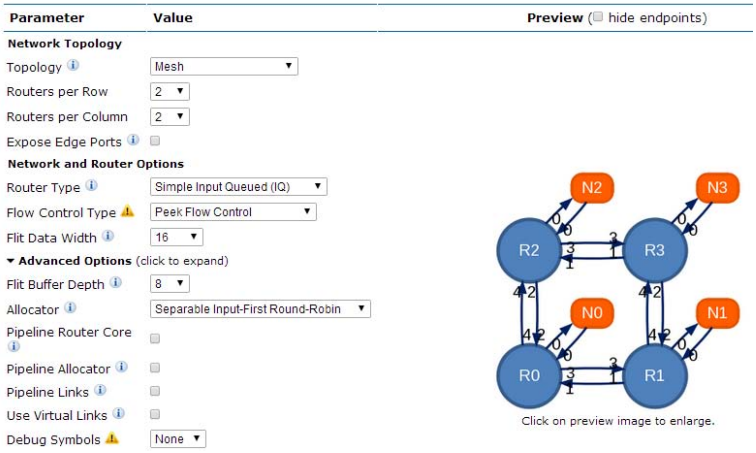


Figure 2.4: CONNECT user interface.

times, the router datapath is pipelined to improve the clock speed. Router complexity is influenced by the number of incoming/outgoing ports and the flow control and routing logic.

2.2 CONNECT NoC generator

In this section, we discuss about the useful features and usage of CONNECT, a web based synthesizable RTL generator for custom Network-on-Chip (NoC). Our choice of CONNECT NoC generator has been motivated by multiple reasons. First, CONNECT can be used for generating a NoC of arbitrary topology and supports a large variety of router and network configurations. Second, CONNECT incorporates a number of features fine-tuned for the FPGA platform as opposed to other popular NoC generators, which are intended for ASIC development. Finally, we wish to devise a completely automated and standardized approach for NoC generation in multi-FPGA scenario. CONNECT is a completely automated tool, for which the FPGA partitioning can be easily integrated and automated using our approach as described in later chapters. The application designs using CONNECT are topology-agnostic, aiding design space exploration and faster prototyping.

Figure 2.4 shows the interface of the CONNECT tool. The user interface provides an option to use the built-in topologies and routing strategies in CONNECT for standard NoC toplogies, such as Mesh, Ring, Torus, Fat tree etc. In addition, the user can also specify custom topology and routing using configuration files in CONNECT. The tool supports only deterministic routing. in which the routing decisions are stored in the routers in the

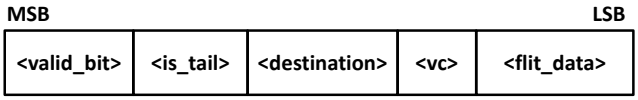


Figure 2.5: Flit format in CONNECT.

form of small look-up tables. The look-up tables have been implemented in a form such that they map efficiently on the distributed RAMs present in the FPGA, which is grossly memory-limited. The routing configuration files in custom NoC are effectively used to specify these look-up table entries. Additionally, the user can specify the router and flow control type, buffer depth and also has the option of using a pipelined router and debug symbols. The network and router configuration (along with some advanced options) shown in Figure 2.4 is the configuration which has been used throughout this work.

In addition to optimize on the on-die storage of FPGA, the CONNECT tool also efficiently exploits the abundance of wire resource on the FPGA by using wide link width, which carry additional flit control information. Figure 2.5 shows the flit format used in CONNECT. Moreover, since the FPGA hardware often operates on a much lower clock frequency as compared to ASIC, a shallow pipeline router is more desirable for a NoC on an FPGA. CONNECT exploits this feature by providing a shallow pipeline router micro-architecture of 1-3 stages, as opposed to the conventional 5 stage pipeline used in an ASIC NoC. In fact, we use a single stage router pipeline throughout this work, and yet, the clock frequency is not limited by the router micro-architecture in our experience. This not only improves the network latency, but also reduces the flip-flop requirement in between the pipeline stages.

For flow control, CONNECT tool has both the mechanisms available ,namely the credit-based flow control and the peek flow control. Use of peek flow control further minimizes the storage requirement in CONNECT. Finally, the user is also free to choose buffer depth in routers and use additional debug symbols for verification purposes. The tool is freely available for use on http://users.ece.cmu.edu/ mpapamic/connect/.

Chapter 3

Automated Network-on-Chip Partitioning

FPGAs have been extensively deployed for full system prototyping and verification of large ASIC designs [15, 25, 33]. Lowering costs, as a consequence of advanced manufacturing processes and increasing competition, coupled with rising operating speeds, have rendered FPGA prototyping a considerable advantage over other known alternatives. These include RTL simulation [13] and virtual prototyping [30], which are known to be slow and expensive in comparison. FPGAs make good use of available fine-grained parallelism and find applications in High Performance Computing (HPC) and hardware acceleration. A number of hardware/software co-simulation tools, such as ProtoFlex [7], have been developed which rely on FPGA for fast hardware emulation of complex designs and also provide testing and debugging support.

Modern day customized chips, on the other hand, have also seen a remarkable rise in design complexity with advanced ASICs consisting of over a billion gates. Although FPGAs have also witnessed a rapid increase in integration costs, resource constraints in terms of limited gate capacity, DSP units, I/O pins, memory arrays and clock speeds render FPGAs almost impractical for prototyping of very large designs using a single chip. To mitigate this issue, the large ASIC designs are often partitioned across multiple FPGAs [26] such that the major functional blocks reside over different FPGAs, which are further provisioned with an appropriate communication interface. Multi-FPGA partitioning is still considered to be one of the toughest challenges in FPGA prototyping. It is cumbersome and error-prone as it requires considerable manual intervention and knowledge of partitioned modules.

To this end, we believe that a standardized and simplified communication interface based on Network-on-chip (NoC), with automated partitioning across multiple FPGAs, could greatly allay the designers from the tedious partitioning process. The partitioned NoC should also be flexible in terms of topology and routing configurations. Since several modern day ASICs and MPSoCs also employ NoCs for communication between modules, our partitioned NoC would also be effective for prototyping and debugging of these designs on multiple FPGAs, which are otherwise limited by the resource constraints of a single FPGA. This approach is also handy for designing multiple FPGA-based hardware accelerators, using the underlying NoC as the communication interface. This would become apparent from the illustration of a FPGA-based Boolean Matrix Vector Multiplication accelerator, whose scalability is determined by the ability to partition its network interface over several FPGAs.

3.1 NoC Partitioning 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, such as in Fat-tree or Projective Geometry-based topologies, which we investigate in this work.

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 that further complicate the design. Further, modules on different FPGAs may 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 propose to 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

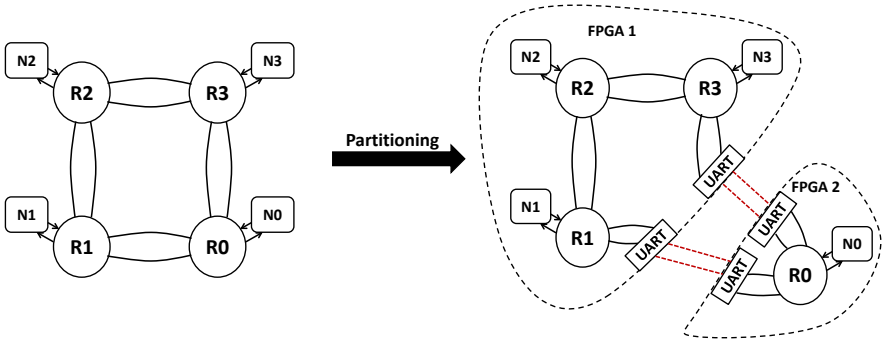


Figure 3.1: Example partitioning of an NoC with four routers on two FPGAs. The router R0 (along with its processing element N0) is mapped onto a separate FPGA. Communication between FPGAs takes place over serial UART interface.

also eliminates the need for a common clock reference. To this effect, the 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 3.1 provides an an example of NoC partitioning of four routers on two FPGAs, with UART as the serial communication interface between the FPGAs. In the next section, we elucidate the details of our Python script which automates the network partitioning.

3.2 Implementation Details

As stated in the previous section, we require asynchronous serial interface for across-the-partition 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 are remote and 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. This is

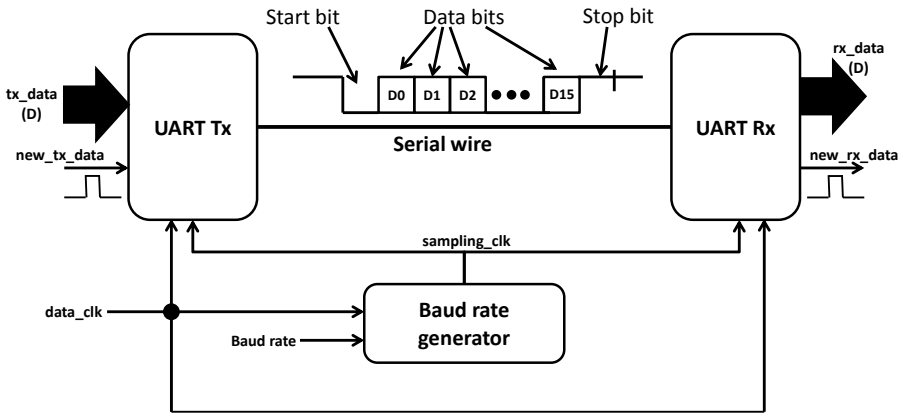


Figure 3.2: Serial transmission using UART.

shown in Figure 3.2. 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. We have implemented separate modules for the baud generator and the UART transmitter and receiver. Our script automatically partitions a given NoC into separate sub-networks which can be inter-connected via this serial interface and is designed for partitioning NoCs generated using the CONNECT tool with the following parameters:

- **Router Type:** Simple Input Queued (IQ)
- **Flow Control Type:** Peek Flow Control
- **Allocator:** Separable Input first Round-Robin

Since flow control signals, along with the serial data links, are also required to cross the FPGA partition, we would like to minimize the number of signals required for flow control 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, which is half the requirement of credit-based flow control.

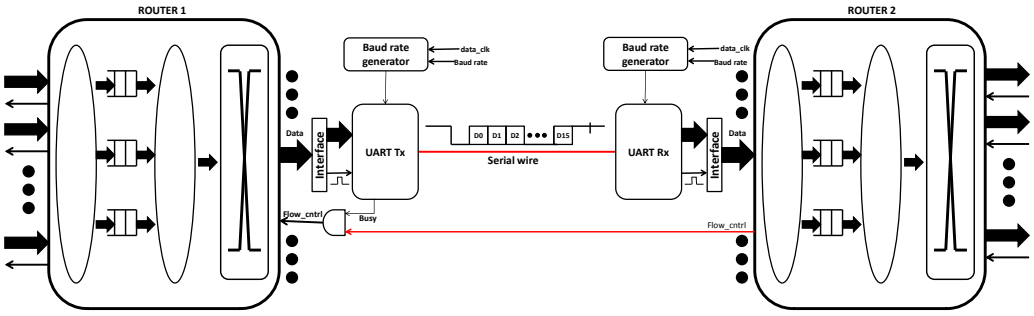


Figure 3.3: Communication between routers using UART. Red lines indicate the wires crossing the interface.

Additionally, peek flow control eliminates the need for maintaining registers for storing credits. We also do not use Virtual Channels, although it helps maximize network performance, for the same reason to minimize the FPGA pin count requirement.

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, as already mentioned in section 2.2, 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 increased). Therefore, the total of the flit data width and the destination address width cannot exceed 16 bits. Similarly, on the receiver side, the bits carrying to the destination address on the receiving router are appropriately connected to the corresponding bits on the received data port of UART receiver. On the transmitter side, the most significant bit of the outgoing flit, 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. Figure 3.3 shows an example of two routers along with the additional UART circuitry.

Our script, built on Python, uses the above approach to partition the network. The usage details of the script is provided in Section 8.2. The script takes the original CONNECT network as input, parses it to infer the interconnects between the routers and then determines the ports which are required to cross the input partition. For those ports, the script inserts additional circuitry for asynchronous serial communication. NoCs partitioned using this script have been tested on simulation, as well as on FPGA.

3.3 Advantages of Automated NoC Partitioning for Multiple FPGAs

In addition to some of the advantages of automated NoC partitioning for multiple FPGAs listed earlier, we enumerate a few more advantages of our approach:

- This approach *scales* seamlessly to a large number of modules and FPGA partitions. The application developer is not required to be aware of the details of the serialization/de-serialization modules used by the NoC. The user modules are also *oblivious* to the NoC partitions and may continue to use the NoC in the same manner as without the partition.

- The partitioning produces a *synthesizable* Network-on-Chip spread across multiple FPGAs and is flexible in terms of network *topology* and *routing*. Since the network is generated using CONNECT modules, it is also optimized for FPGA fabric.

- This NoC-based partitioning approach allows several modules operating at different clock speeds to effortlessly communicate with each other. This is because, the partitioned network uses asynchronous serial links, instead of synchronous parallel wires, to communicate with adjacent routers. In fact, our approach could be used in prototyping voltage-frequency islands for *GALS*-based networks-on-chip onto FPGAs, as described in [21]. The authors have demonstrated 40% energy savings using this approach on real video applications.

- Since the routers use serial links to communicate across different FPGA partitions, this approach minimizes the requirement of sparsely available *I/O pins* on the FPGA. Use of peek flow control, instead of credit-based flow control, further helps in this regard.

- This approach enforces *modularity* and *information hiding* in a multi-FPGA design, further improving design productivity and development time of multi-FPGA designs.

- Our approach also allows for *design-space exploration* of NoCs over multiple-FPGAs. A network topology could have significant bearing on the overall performance and design space exploration would help strike the right balance between cost and performance.
- Our approach can be extended to use *high-speed* interconnect technology, such as RapidIO [12, 27], instead of UART links to achieve higher performance. This is a work in progress.

Chapter 4

Boolean Matrix Vector Multiplication

4.1 Introduction

Matrix-vector multiplication is central to a large number of scientific computing applications. Boolean matrix vector multiplication is defined over the smallest finite field GF(2). A finite field is an arithmetic system, consisting of finite number of elements (known as the *order*), over which commutative addition, subtraction, multiplication and division operations have been defined. Equation 4.1 defines the boolean matrix vector product of an *n*x*n* boolean matrix *A* with a boolean *n*-vector *v*.

$$A.v = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} (a_{1,1} \wedge v_1) \oplus (a_{1,2} \wedge v_2) \oplus \cdots (a_{1,n} \wedge v_n) \\ (a_{2,1} \wedge v_1) \oplus (a_{2,2} \wedge v_2) \oplus \cdots (a_{2,n} \wedge v_n) \\ \vdots \\ (a_{n,1} \wedge v_1) \oplus (a_{n,2} \wedge v_2) \oplus \cdots (a_{n,n} \wedge v_n) \end{pmatrix} \tag{4.1}$$

Boolean matrix vector product has important applications in coding theory [3], cryptanalysis [19] and image compression [28], among many others. Several secure systems today rely on the computational intractability of factoring large numbers [31]. Lenstra et. al [19], proposed Number Field Sieve (NFS), perhaps the fastest known algorithm till date for factoring large integers. NFS, on practical applications, would still require several months of computational time to be completely solved on several computing clusters. There are four major steps in the NFS algorithm:

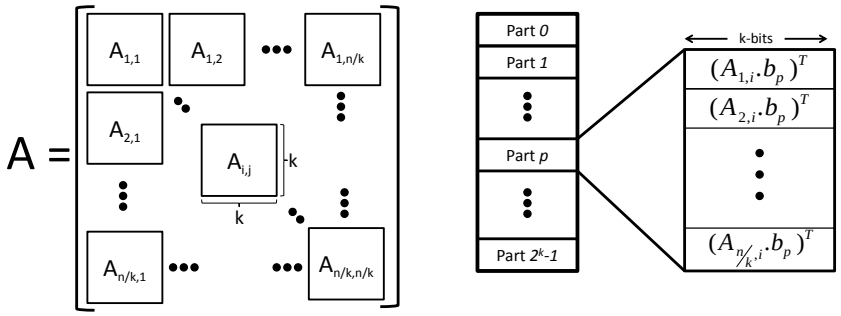
1. Polynomial selection
2. Sieving
3. Matrix Step
4. Square root

Sieving and *Matrix Step* are considered to be the most time-consuming steps in the NFS [1]. In particular, the Matrix Step involves finding linear dependencies in a large boolean matrix. Block Wiedemann [8] algorithm is often used for this purpose. This involves repeated multiplying of a very large boolean matrix A (matrix size in the range $10^6 - 10^{11}$), starting with a random boolean vector v_i resulting in product vectors $A.v_i, A^2.v_i, ..., A^r.v_i$. This is done for 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".

In general, the boolean matrix-vector multiplication (BMVM) is considered to be quadratic w.r.t. to the size (column/row size) of the matrix A. However, since we are repeatedly using the matrix A for computing products with random vectors over thousands of iterations, any improvement in the computation time for computing product by exploiting the structure of matrix A, would directly reflect in the runtime of the *Block Wiedemann* algorithm, and therefore, also of *NFS*. Geiselmann et. al [14] have made certain assumptions on the *sparsity* of matrix A, and exploited it for accelerating the boolean matrix-vector product using custom hardware. More details on this work are provided in Chapter 6.

In this work, we do not wish to make any assumptions on the sparseness of matrix A as in [14], since we want our technique to be more generally applicable to other applications. We do, however, believe that exploiting the sparsity structure of matrix would be necessary for scalability of our technique, and briefly discuss it in Chapter 6. We have left these ideas as a part of future work. Our technique is based on the recently proposed combinatorial approach for matrix vector multiplication by Ryan Willams [32]. This approach allows matrix-vector multiplication to be performed sub-quadratically in $O((n/k)^2)$ on a $log(n)$-word RAM, with a single pre-processing step of $O(n^{2+\varepsilon})$, when $k = \varepsilon.log(n)$.

We describe this technique in section 4.2 (for more details on the algorithm, refer to [32]), and then report our hardware implementation details in section 4.3.



(a) Partitioning matrix A (b) Structure of LUT_i

Figure 4.1: Preprocessing of Matrix A

4.2 Sub-quadratic Method to Matrix Vector Multiplication

We now describe the algorithm for Boolean Matrix Vector Multiplication, which allows for sub-quadratic multiplication, with some pre-processing. The algorithm is based on [32], and has two phases: (i) pre-processing and (ii) computation phase. The pre-processing step uses the given input nxn matrix A to build look-up tables. The computation step uses the generated look-up tables to perform the actual multiplication with the given input vector v. We now individually describe the steps involved in the two phases. Note that we use a modified convention over [32] for simplified understanding.

Pre-processing Phase:

1. Partition the input matrix A into blocks (tiles) of size k x k bits as shown in Figure 4.1, for a given user defined parameter k. The value of this parameter k has important ramifications on the requirements of the number of look-up tables and computing nodes, size of the look-up tables and the number of operations during pre-processing and computation phases. These relations would become clear by the end of this section.

2. Construct n/k loop-up tables (LUTs), labelled as $LUT_1, LUT_2, ..., LUT_{n/k}$, each of which further consists of 2^k partitions. These partitions are indexed from 0 to $2^k - 1$. Each partition consists of n/k words of size k bits.

3. Populate the LUTs such that the p-th partition of LUT_i consists of k-bit words cor-

responding to values $A_{1,i}.b_p, A_{2,i}.b_p, ..., A_{n/k,i}.b_p$. Here b_p is a k-bit vector representation corresponding to the decimal value p. This means that b_0 is an *all-zero* vector and b_{2^k-1} is an *all-one* vector. The structure of the LUT_i is also shown in Figure 4.1.

Note that pre-processing step is equivalent to pre-computing and storing all possible products of the blocks of matrix A, i.e. for $A_{1,1}, A_{1,2}..A_{n/k,n/k}$. Since there are 2^k such possible products for each block, and n^2/k^2 blocks, a total of $2^k.(n^2/k^2)$ matrix products are required during pre-processing. Since each of this product requires $O(k^2)$ computational operations, and hence, the total pre-processing can be achieved in $O(n^2.2^k)$ operations. Also if the vector v and the product vector $v' = A.v$ are also partitioned into n/k sub-vectors, each of size k-bits, as $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n/k} \end{pmatrix}$ and $v' = \begin{pmatrix} v'_1 \\ v'_2 \\ \vdots \\ v'_{n/k} \end{pmatrix}$, then the sub-vector v'_i can also be expressed as:

$$v'_i = (A_{i,1}.v_1) \oplus (A_{i,2}.v_2) \oplus ...(A_{i,n/k}.v_{n/k}) \tag{4.2}$$

In equation 4.2, the operator $\oplus$ performs bit-wise *xor*-operation on the k-bit partial products. This is the essence of the computing phase described next.

Computing Phase:

1. Partition the input vector v into n/k k-bit sub-vectors $v_1, v_2, .., v_{n/k}$, s.t. $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n/k} \end{pmatrix}$. Let v' be another n-bit vector. Partition v' into k-bit sub-vectors in a similar manner, as $v' = \begin{pmatrix} v'_1 \\ v'_2 \\ \vdots \\ v'_{n/k} \end{pmatrix}$, and initialize it to 0.

2. Look-up the partition p_i in the look-up table LUT_i (obtained from pre-processing) corresponding to the sub-vector v_i for all i in $\{1, 2, .., n/k\}$.

3. For the j-th (j ranging from 1 to n/k) word e_j in partition p_i, update v'_j as bit-wise *xor* of current v'_j with e_j. Do this for all partitions p_i, obtained in step 2.

4. The product $A.v$ is then given by $v' = \begin{pmatrix} v'_1 \\ v'_2 \\ \vdots \\ v'_{n/k} \end{pmatrix}$.

It can be inferred easily that the computing phase requires n^2/k^2 bit-wise *xor*-operations of size k-bits. The matrix-vector multiplication can, therefore, be computed in $O(n^2/k^2)$. In [32], the authors have used $k = \varepsilon log_2(n)$, with $\varepsilon < 0.5$, so that the multiplication can be carried out in $O(n^2/(\varepsilon log(n))^2)$ on a $log(n)$-word RAM.

We summarize the important analysis related to the algorithm, when used with parameter k here:
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

It is observable that increasing the value of k would reduce the computation time due to the reduced number of operations, but would, in turn, also increase the memory requirement (in LUTs) and the pre-processing time (occurring in $O(2^k.(n^2/k^2))$) exponentially. A careful selection of the parameter k is thus required for achieving the right balance between the three. Also note that the extreme case when $k = n$ requires a single LUT storing n-bit product vectors corresponding to all possible 2^n combinations of input vector. Any multiplication would only require a single look-up into the corresponding entry. However, this is exponential in memory and pre-processing time and cannot be applied to matrices exceeding a few 10s of bits in size.

4.3 Implementation Details

In this section, we shall describe the implementation details of the algorithm described in Section 4.2. The goal of our implementation is to have a *flexible* and *scalable* software library framework for FPGA-accelerated iterative boolean matrix multiplication using the above algorithm.

4.3.1 Pre-processing

As described in section 4.2, the algorithm is composed of two phases: the pre-processing phase and the computation phase. Since the pre-processing phase is required only once for a given input matrix, which is assumed to be re-used multiple times over several iterations in the target applications, we apply this phase completely on the software end using MATLAB. The source code snippet, along with usage details, for this step in available in section 8.1. The script takes the matrix size n, ε(s.t. $k = \varepsilon.log(n)$) and folding factor as an input parameter. We describe the concept of "folding", which is used by the folding factor, later in this section.

It must the noted that although the authors in [32] use $\varepsilon < 0.5$ for reasons mentioned already, we make no such assumption and leave it to the user to best determine the appropriate value of ε. The code produces several '.txt' and '.data' files as output, corresponding to the look-up tables used in the FPGA and software implementation of the algorithm, respectively and having the same structure as described in Figure 4.1(b). The $LUTs$ are generated by multiplying each blocks (tiles) of size kxk of the input matrix A by all possible vector combinations of size k, and storing the output values. In the current code, the matrix A, of size nxn, is randomly generated with uniform probability of $1's$ and $0's$. The matrix size if also assumed to be a power of 2, which can be easily modified to make it more general.

4.3.2 Hardware implementation

We now start discussing the FPGA implementation of the algorithm. As evident in section 4.2, the algorithm is memory-intensive, rather than computation-intensive, due to the large sizes of the look-up-tables. To preserve data locality and improve computation time, we would like the $LUTs$ to completely reside on the FPGA hardware. In order to best utilize the available memory resources in the FPGA, we first observe that modern FPGAs are provisioned with abundant Block RAM (BRAM) resources. For instance, *Xilinx Vitex 6* (used in this work) has a large number of 36Kb Block RAMs, totalling upto 38Mb of internal BRAM storage. The internal structure of a typical FPGA is shown in Figure 4.2. It can be observed that the Block RAMs are broadly distributed over the entire FPGA fabric.

As mentioned earlier, the memory requirement of a single LUT is $(n/k) * 2^k$ words (with each word storing k bits). We would ideally like the $LUTs$ to be implemented automatically using Block RAMs. In order to achieve this, we use the following template for an LUT:

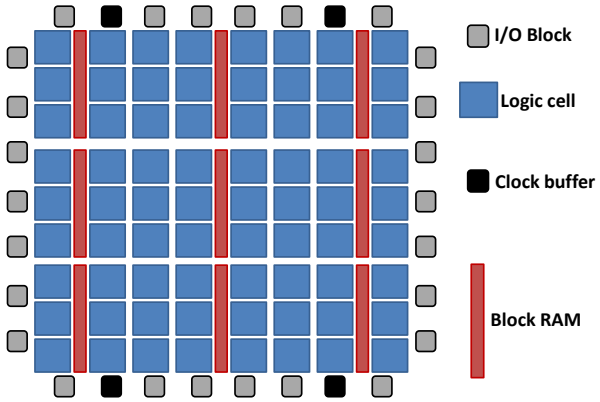


Figure 4.2: FPGA internal structure.

```
module LUT( clk, write_enable, address, input_data,
  output_data);
.
.
.
  parameter file_name = "preprocessed_1.data";
  reg [RAM_WIDTH-1:0] RAM_1R1W [(2**RAM_ADDR_BITS)-1:0];

  initial
     $readmemb(file_name, RAM_1R1W , 0, (2**RAM_ADDR_BITS)-1);

  always @(posedge clk) begin
  output_data <= RAM_1R1W[address];
        if (write_enable)
          RAM_1R1W[address] <= input_data;
  end
.
.
.
endmodule
```

This allows the (Xilinx) synthesiser tool to automatically infer Block RAMs for the *LUTs*. Also note that the code uses *$readmeb* command to read and initialize the *LUTs*

directly from the pre-processed input files (provided using parameter in this module), so that the initialization is not required.

It must be noted that the loop-up tables, implemented in the form of Block RAMs are required to “exchange information" based the algorithm described in Section 4.2. The BRAMs could be situated far off from each other on an FPGA, or even reside on different FPGAs altogether, in a multi-FPGA implementation (memory constraint of FPGA limits the size and number of LUTs that can reside on a single FPGA). Network-on-chip would not only help in seamless communication between the LUTs, but would also help improve the clock speed. In our FPGA implementation, with each of the n/k $LUTs$ obtained from pre-processing, we associate a “computing node". These nodes can communicate with each other using k-bit messages over an NoC and these nodes are indexed from 1 to n/k. A compute node i also maintains a k-bit register v'_i, initialized to 0. A compute node indexed i would look-up the partition p_i in the look-up table LUT_i corresponding to the sub-vector v_i. The compute node i sends (n/k) k-bit values in the partition p_i to the corresponding compute nodes. Every compute node also would receive n/k such k-bit messages. The register v'_i corresponding to node i is computed by bit-wise xor of all the messages received by node i. The product $A.v$ is then given by $v' = \begin{pmatrix} v'_1 \\ v'_2 \\ \vdots \\ v'_{n/k} \end{pmatrix}$.

We use a NoC generated by CONNECT[23] for our implementation in which the computing nodes are connected as the “processing elements" (PEs) at the input/output ports of the generated NoC. These nodes use the interface provided by CONNECT to exchange messages. 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

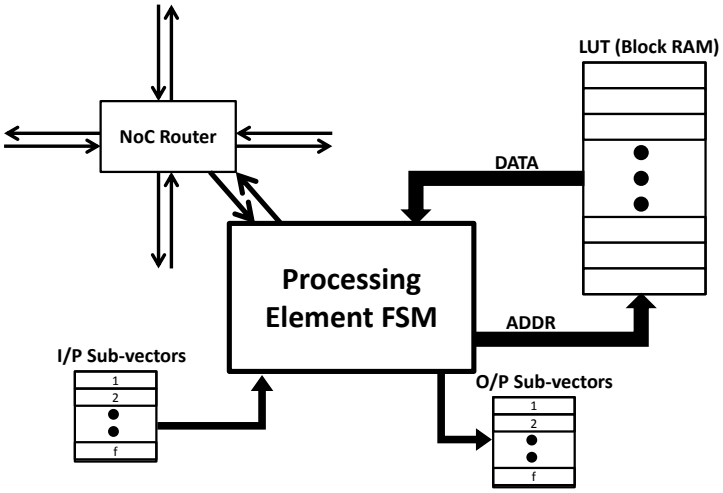


Figure 4.3: Processing element.

- **Allocator:** Separable Input first Round-Robin

Since the number of computing nodes required grows linearly with the matrix size (as n/k), the number of ports required for the NoC would also grow linearly. This could pose and exorbitant requirement on the number of NoC ports since most topologies do not scale well to a very large number of endpoints. To this end, we implement “folding" of computing nodes, which allows multiple processing elements connected to a single endpoint of an NoC to serve as multiple computing nodes. The extent of such folding, referred to as the “folding factor" f, is a user-specified parameter in our implementation. Our folding follows a cyclic assignment, i.e. an instance with n computing nodes and n/f processing elements or endpoints, would assign computing nodes $1, f+1, 2f+1, ..., (n-f)+1$ computing nodes to the processing element 1. Since a single processing element is now required to send and receive f^2 times the number of messages to the case where there is no folding with same number of processing elements (or ports in the network), the message traffic would also increase, resulting in higher network utilization. Since only one message can be sent/received on a single NoC port, the folded design would exploit lower concurrency, thereby lowering performance. The folding parameter is also specified while generating the LUTs in the pre-processing step, which results in large memory sizes compared to the case of no folding. The caveat of large folding, however, is that large LUT could also lower the maximum operational frequency of the design, thereby further lowering its overall performance. These effects are discussed in greater detail in Chapter 5. Figure 4.3 shows a single processing element (multiplication node) with folding factor f

in our implementation.

Since a number of our target applications require computation of several iterations of boolean vector multiplications, *viz.* $A.v, A^2.v, ..., A^k.v$, given an input vector v, we provision our processing elements to support iterative matrix vector product. The number of such iterations is again provided as parameter by the user. Support for iterative multiplication would require the processing elements to perform multiplication of matrix A with input vector v'_k in the kth iteration, where v'_k is the resulting product vector after the $(k-1)th$ iteration. Since the resulting sub-vectors v'_k are local to the processing elements in the kth iteration, this task seems trivial. However, iterative multiplication poses a unique challenge of *synchronization* of processing nodes after every iteration. This means that a processing element should only commence the kth iteration after ensuring that all other processing elements are also ready for starting kth iteration. If this is not the case and a processing element (which has already sent and received all messages of previous iteration) jumps to next iteration before others, the messages sent by this PE could be misinterpreted by other PEs to belong to previous iteration, resulting in incorrect computation of product values. One solution could be to use additional bits in messages to indicate the *id* for the iteration to which the message belongs. This would allow PEs to proceed to next iteration without waiting for others but this is expensive as the flit width is limited and this would also require buffering of incoming messages of future iterations. Other option could also be to use special messages to broadcast information to all PEs whenever a PE has completed an iteration. Each PE would maintain a list of completed PEs and would proceed to next iteration only after ensuring all others are also ready to proceed. While this seems more feasible, it also has performance penalty associated with the broadcast messages after every iteration.

In our approach, we observe that the behaviour of the network after in each iteration would be exactly identical if the PEs are exactly synchronized. Therefore, if we know the maximum number of cycles required for a PE to complete an iteration, all PEs could wait to synchronize at this cycle count after every iteration, thereby eliminating the need for additional broadcast messages. It is important to note that this count is dependent on the topology and routing policy used. We specify the count as a parameter to the processing element in our design, which could be determined by the user by simulating one iteration of the multiplication on the software. Our processing elements display the cycle counts at which the processing elements are done with one iteration (when they have sent and received all messages pertaining to that iteration), and the maximum of this count can be specified in that parameter. The source code snippet of our processing element is included in 8.3. Other parameters used by this module are also specified and explained

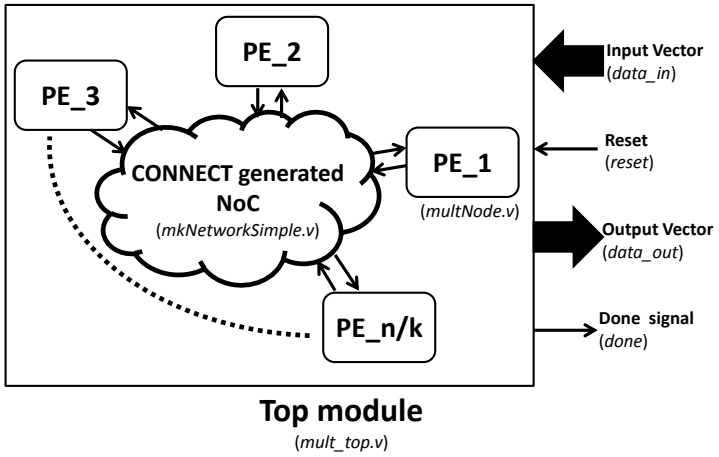


Figure 4.4: Top module for Boolean Matrix Vector Multiplication (BMVM).

in 8.3. Figure 4.4 shows the top level user module along with their module names in our design, consisting of several instances of processing elements connected by a CONNECT generated NoC. It must be noted that the NoC can itself be partitioned using our script, in which case the top module would consist of multiple instances for NoC corresponding to different partitions, with serial wire and flow control signals between them. We have only tested our partitioned NoC for boolean vector multiplication on a single FPGA, with wires corresponding to different partitions connected internally. Since the partitioned NoC itself has been tested successfully on separate Virtex 5 boards, we believe that multiplier design should also work for multi-FPGA case. The interface to the top module is kept as simple as providing the input vector followed by a reset signal as input, and obtaining the output vector on the done signal at the output.

In our final step, we wish to encapsulate our multiplication hardware within a software library to make it usable for target applications, mostly running on software. For this purpose, we use Reusable Integration Framework for FPGA Accelerators (RIFFA 2.0) [16], which provides a simple open-source integration framework for FPGA-CPU communication over high-speed PCIe links. On the software end, RIFFA provides a simple API to open/close an FPGA connection, send/receive data from FPGA and to send reset the FPGA user module. Although we use a single channel (corresponding to network sockets and parallel user cores), RIFFA can support upto 12 channels. The data transfers in RIFFA occur through PCIe transactions, with FPGA acting as the DMA bus master. For more details, the reader can refer to [16].

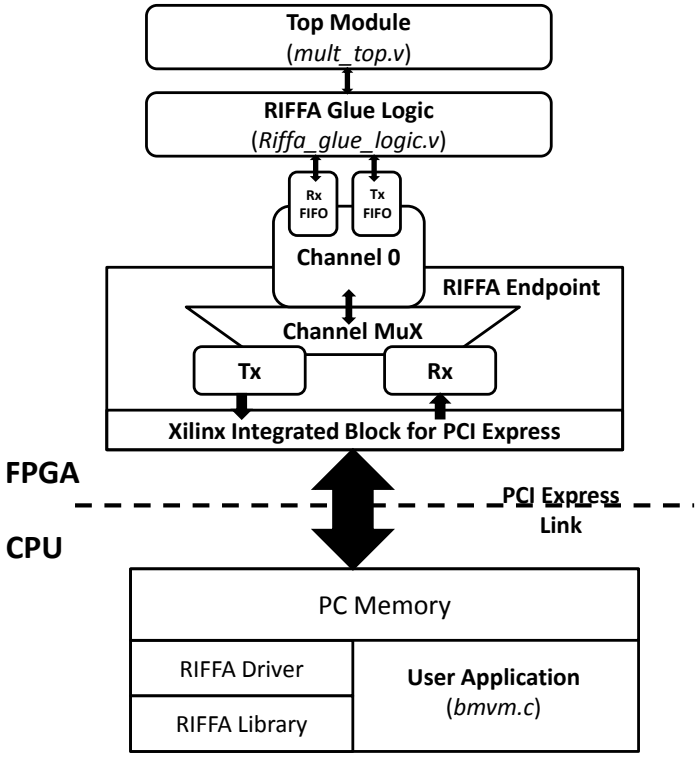


Figure 4.5: BMVM framework with RIFFA.

We demonstrate the usage using a simple C application. The application resets the FPGA and sends an input vector, on which the multiplication is performed by the FPGA, before waiting to receive the output product. Since the application is written for a 32-bit processor, appropriate precautions are taken to make it compatible with the *little endianess* of the RIFFA hardware modules supporting 64-bits. We also implement the necessary glue logic for the hardware interface between RIFFA endpoint and multiplier module as shown in Figure 4.5.

4.3.3 Software implementation

For comparison, we have also implemented a multi-threaded software implementation of the described boolean matrix vector multiplication (BMVM). The source code is available in 8.4. A shared memory model, using the *pthread* library is used , in which a thread

is an exact equivalent of the processing element in the FPGA. The input parameters to the application include the input matrix size, number of parallel threads, size of look-up per vector (sub-vector size to the power of 2), folding factor and the number of iterations. The threads are first initialized with look-up tables, generated using the MATLAB code as described earlier in the section, and are also provided with the input sub-vector. Since multiple parallel threads may try to update the same sub-vector at once leading to concurrency issues described earlier, mutual exclusion between parallel updates is ensured using *locks*. All the threads are also required to synchronize on a *barrier*, before proceeding to the next iteration. Our implementation, in its current form, is naive and has scope for improvement. For instance, an entire 32-bit integer is allocated for a single bit in a matrix/vector, although bit-level operations are also possible in most processors. However, even with these software level optimizations, we do not expect the application speedup by more than a few factors.

Chapter 5

Experimental Evaluation

5.1 Setup

Pre-processing

Our pre-processing script has been written in MATLAB and tested on MATLAB version R2009b.

Hardware BMVM

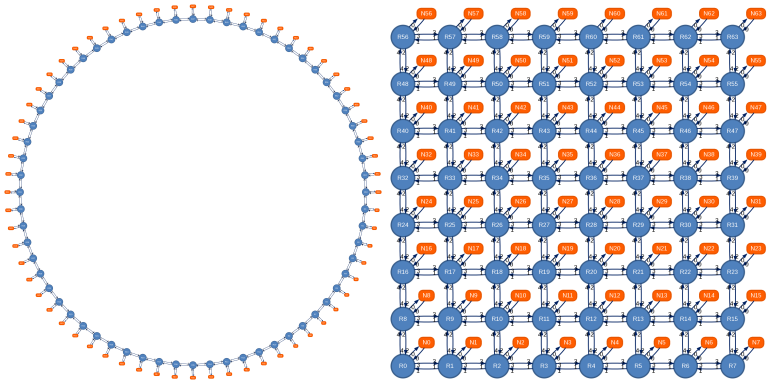
We used Icarus Verilog Simulator (version 0.9.4) for software testing and debugging of our verilog files. The hardware was configured on Xilinx Virtex 6 ML605 XC6VLX240T-1FFG1156 Evaluation Board. We used Xilinx ISE version 14.3 for synthesis and generation of our hardware bitfiles in all our experiments. We used RIFFA 2.0 for interfacing the hardware and software over PCIe, and gcc compiler (version 4.8.2) for compiling the software endpoint on a Intel i7-2600 processor, operating at 1.6GHz.

Software BMVM

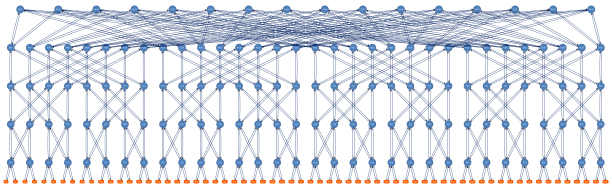
For multi-threaded software BMVM, we use g++ compiler (version 4.4.7) on a six core Intel Xeon E5-2620 processor (Model 45, 15360 KB cache), operating at 1.20 GHz. Each processor can simultaneously execute two threads.

5.2 Results and Comparisons

In order to evaluate the impact of topology, folding, matrix size and number of iterations on the execution time of the algorithm and the FPGA modules, we evaluate our hardware implementation on four different topologies, namely Ring, Mesh, Torus and Fat Tree, for



(a) Ring (b) Mesh



(c) Fat Tree

Figure 5.1: Network topologies (generated by CONNECT tool) with 64 PEs as used configurations 1 and 2.

two different parameter configurations ($n = 512, k = 4, f = 2$ and $n = 1024, k = 4, f = 4$). Each topology has 64 processing elements. Figure 5.1 shows the Ring, Mesh and Fat Tree topologies (Torus looks identical to Mesh, but with end routers connected in a wrap-around fashion). All our user modules were made to operate at 100 MHz clock frequency. We also partition a small Mesh network with four PEs (with 4 PEs, Mesh is identical to Ring and Torus) using our script and provide the results for the configuration $n = 64, k = 8, f = 2$. Table 5.2- 5.6 show the comparative results, with the multi-threaded software BMVM as the baseline for speedup, for the three configurations. The software time only excludes the initialization time for the look-up tables and only includes the computation time in the parallel region of the code. Table 5.7 shows the resource utilization for the Fat tree topology for configuration 2.

Configuration 1: $n = 512, k = 4, f = 2$

	Ring	Mesh	Torus	Fat Tree
Max Cycles	4110	840	770	680
Real Giga-BOpS	0.398	1.95	2.13	2.40
Virtual Giga-BOpS	6.38	31.2	34.0	38.6

Table 5.1: Cycle count and performance in terms of giga bit operations per second assuming 100MHz clock for a single multiplication for configuration 1.

Iterations	Time (in msec)					Speedup (Norm. to Software)			
	Software	Ring	Mesh	Torus	Fat_tree	Ring	Mesh	Torus	Fat tree
1	2.70	0.076	0.054	0.052	0.050	35.5	79.4	51.9	54
10	16.90	0.484	0.118	0.112	0.098	34.9	142.5	150.2	172.4
100	160.7	4.30	0.922	0.852	0.716	37.4	174.3	188.6	224.4
1000	1636.5	42.23	8.49	7.78	6.88	38.7	192.8	210.3	237.9

Table 5.2: Comparative results for configuration 1 (average over 100 experiments).

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

	Ring	Mesh	Torus	Fat Tree
Max Cycles	15990	3540	2790	2250
Real Giga-BOpS	0.41	1.85	2.35	2.91
Virtual Giga-BOpS	6.6	29.6	37.6	46.6

Table 5.3: Cycle count and performance in terms of giga bit operations per second assuming 100MHz clock for a single multiplication for configuration 2.

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

Configuration 3: $n = 64, k = 8, f = 2$

	Mesh	Partitioned Mesh
Max Cycles	40	138900
Real Giga-BOpS	1.6	0.00046
Virtual Giga-BOpS	102	0.029

Table 5.5: Cycle count and performance in terms of giga bit operations per second assuming 100MHz clock for a single multiplication for configuration 3. Partitioned Mesh assumes a UART baud rate of 115200 bps.

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 5.6: Comparative results for configuration 3 (average over 100 experiments).

Resource	Number Used	% Utilization
Slice Registers	28,794	9 %
IOBs	1	1 %
Slice LUTs	50,482	33 %
Block RAMs	144	34 %

Table 5.7: Resource utilization for Fat-tree topology for configuration 2.

5.3 Discussion

Table 5.7 shows the resource utilization for Fat tree topology (the most resource-hungry topology) for configuration 2. As evident from the numbers, our implementation makes efficient use of available Block RAMs for generating look-up tables, as desired. This was also verified in the message console of the Xilinx ISE, which indicated how the look-up tables were mapped to Block RAMs. Tables 5.2 and 5.4 also indicate that topology can

play a substantive role in determining the clock cycles for multiplication, since our algorithm is communication-intensive and not computation-intensive. The ring topology is a low-radix topology and which gets quickly congested in comparison to the high-radix fat tree topology. In fact, our results show a conspicuous correlation between network cost and performance (the cost increases moving from ring to mesh to torus to fat tree but performance also improves accordingly). Typically, any massively parallel implementation of boolean matrix vector multiplication requires significant communication [2], but using NoC can help in striking the right balance between cost and performance.

The execution time for multiple iterations should grow linearly with the number of iterations. In software BMVM, the additional time for thread creation and join inevitably gets included in the execution time measured, which is why the total software execution time seems to increase super linearly for smaller iterations. The execution time of software was also observed to show higher variance over 100 experiments, since the software BMVM has several synchronization calls. For hardware BMVM, since the execution time also includes the round-trip time incurred in sending and receiving the vector from the FPGA over the RIFFA interface. The effect of the roundtrip time is dominant in low iterations (1-10), but its effect is minimal in larger iterations (100-1000), where the actual computation time on FPGA dominates the total time. Configuration 1 and 2 differ in matrix size and folding factor on the same topology. Although the number of messages quadruples with doubling of the folding factor, the cycle counts in Table 5.4 are lower than 4X the cycles counts in Table 5.2. This is because, with increase in network traffic, the overall network utilization has also improved. This, to an extent, highlights another benefit of folding, which results in sub-quadratic increase in execution time on same topology.

As far as the benefits of the algorithm are concerned, the total number of operations in a single BMVM have been reduced by a factor of k^2, ascribing to the pre-processing of the input matrix. This is also perceptible in our results in Table 5.2 and 5.4, in which the real performance, indicating the number of actual bit-level operations per second performed by the hardware, is lower than the virtual performance, which also includes the operations in the boolean matrix vector multiplications which get “hidden" due to pre-processing for the obtained cycle counts to calculate performance. As a result, the pre-processing helps to attain a “virtual" performance, which is much harder to achieve otherwise with the limited availability of hardware. In effect, pre-processing trades of performance with memory (required to store pre-processed data). We also exploit available parallelism, such that upto (n/fk) k-bit operations can be performed concurrently. Therefore, upto n/f actual bit level operations can be performed every cycle, that would lead to full utilization of the bit operators on the hardware. Note that n/f is constant for configuration 1 and 2, but

the performance is higher in configuration 2 as a result of better network utilization, as discussed earlier, due to folding. Although not equal, Fat tree topology can achieve close performance to this theoretical bound. Our hardware has scope for further optimizations to approach this bound.

Although our multi-threaded software BMVM is efficient in comparison to a naive single-threaded implementation, there is a glaring difference between the FPGA and software performance, which can be attributed to several factors. Since the algorithm is communication-intensive and the software BMVM uses shared-memory model using *pthread* library, a significant portion of the execution time is spent in acquiring and waiting for locks and on barriers. This needs to be verified in greater details using profiling. With message passing on NoC in the FPGA, a processing element can eject atmost one flit from its router, and the serialization and concurrency issues are automatically handled. FPGA make better use of available fain grain parallelism in BMVM than software, where the number of processing cores is outnumbered by the number of parallel threads. Also, as previously described, the FPGA implementation exploits memory bandwidth efficiently with several distributed Block RAMs available. The FPGAs are also known to be better suited for bit-level manipulations, such as those required for our BMVM algorithm, which further contributes to its speedup.

Finally, as a “proof of concept", we have also tested boolean matrix vector multiplication for a partitioned 2x2 mesh NoC for configuration 3. Table 5.6 and Table 5.5 show the cycle count and speedup results. As is observed, the UART baud rate (115200 bps) is the performance bottleneck in our results, leading to large slowdown. We hope to use faster serial interface in future. It must also be noted, that increasing k can lead to higher virtual BOpS, as observed for configuration 3, since more operations are "hidden", but also require large memory for storage of LUTs.

Chapter 6

Related Work

6.1 Network-on-Chip

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 custom topology, routing, micro-architecture, power, placement, verification among several more. 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 [22] 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 [26] suggested a genetic algorithm based partitioning method for multiple FPGAs within resource and timing constraints. Papamichael et. al [23] developed an automated NoC generator tool tailor-made for FPGAs. Liu et. al [20] 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. We are not aware of any prior work which extends the NoC methodology itself in the multi-FPGA scenario, similar to our work.

6.2 Boolean Matrix Vector Multiplication

Boolean Matrix Vector Multiplication has also been a topic of interest, especially with its application in Number Field Sieve (NFS) for integer factorization [19]. Bernstein [4] proposed a circuit-based implementation of the matrix-step in the Number Field Sieve. Geiselmann et. al [14] proposed a hardware implementation of the "mesh routing" algorithm using several interconnected ASICs. Bajracharya et. al [2] proposed an implemen-

tation of the above mesh routing circuit on a reconfigurable architecture. In mesh routing, a matrix A of size nxn, requires a mesh of size mxm nodes, where $m = \sqrt{n.h}$, h being the maximum number of non-zero bits in any column of matrix A. The mxm nodes are further divided into D blocks, each of size hxh, 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.

Our work offers a different perspective to the BMVM in Matrix Step, which has largely been focussed around the mesh routing, and differs from it in several ways. First, mesh routing has been designed to handle extremely sparse matrices, but our method is better suited for handling dense matrices. This is because, for dense matrices, mesh routing is required to store $O(n^2)$ and preform $O(n^2)$ operations. In contrast, our method requires n/k or $O(n)$ processing elements, and $O(n^2/k^2)$ k-bit operations. Our pre-processing stage significantly helps in reducing the number of operations per iteration of BMVM, making it more suitable to handle iterative BMVM in case of dense matrices. We are yet to modify our algorithm for the case of “sparse" matrices. Second, each message in mesh routing corresponds to a single bit operation, while a message in our technique correspond to k-bit operations. Third, mesh routing has been specifically devised for mesh-based networks, with a tailor-made routing strategy. Our technique, on the other hand, is applicable to a flexible NoC-based interconnection, and can be used with low-radix, as well as high-radix topologies. As our results observe, the chosen topology can have a significant bearing on the overall performance, and this flexibility allows the designer to obtain a fine balance between cost and performance.

There has also been a rich history of literature on accelerating sparse matrix vector multiplication using a multitude of techniques, such as graph partitioning etc, but from the purview of parallel multi-processors and not custom hardware [6, 29].

Chapter 7

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 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 achieves several orders of speedup compared to its software counterpart, in spite of running on a much lower clock frequency. Our preliminary investigations make a case for NoC-based FPGA designs in high-performance and low power computing. This work opens up avenues for research and lays a strong foundation for further exploration of some pertinent ideas, which we discuss next.

A particularly interesting and essential research direction is to rejig the algorithm for specifically handling sparse matrix vector multiplication. This is relevant since a number of applications mentioned in this work use extremely sparse and large matrices. In the current implementation, the technique would not scale to such large matrices. Exploiting the sparsity or matrices could help the algorithm to scale in two ways. *First*, sparsity could be exploited for compression of very large matrices into look-up tables of reasonable size during pre-processing. Previous work on compression of large boolean matrices includes [17]. *Second*, sparse matrices could also help improve performance by reducing network traffic. For instance, a frequently occurring case of *all-zero* message transmission in sparse matrices can be completely avoided, as it does not affect the values at the receiving processing elements. Since pre-processing of large and sparse matrices (which is super-quadratic w.r.t. size of matrix) could itself be very time-consuming, appropri-

ate speedup methods could be required in this direction, perhaps even utilizing FPGA for acceleration. Parallelization of the pre-processing step is another alternative. Also, assigning appropriate values for ε or k is an independent decision problem. Ways to extend this technique for higher finite semirings beyond Galois Field (GF-2) also remains to be investigated. Williams [32] has also briefly discussed about it in his work. It may also be useful if the *max_cycles* for an iteration could be automatically determined at the end of first BMVM iteration and re-used for future iterations, to eliminate the need for manual intervention in this case. Further software optimizations in the *C* application and a GPU implementation of the parallel BMVM algorithm for more meaningful comparison is also required.

The results have established that the topology and partitions have a strong influence on the performance of the application. Another independent problem is that of application-specific topology exploration and finding the "optimum" partition for a given network topology within the resource constraints to maximize the overall performance or network utilization. Finally, some engineering effort to replace the slow UART links with high-speed serial links, such as RapidIO [12] and pipelined access of LUTs is also required to further improve the current performance. Possibility of an ASIC implementation of our hardware can also be investigated in future.

Chapter 8

Appendix and Usage Guidelines

8.1 Pre-processing

Parameters:

- *matrix_size*: Column size (n) of the matrix A.
- *epsilon*: This parameter sets the parameter $k = epsilon.log(n)$
- *folding_factor*: Folding factor f in the BMVM algorithm.
- A: This script generates a random matrix A, but can be provided with A as input parameter as well.
- *dir*: Directory in which the pre-processed files will be created.

Usage: MATLAB script
Output: Pre-processed text and data files in the directory *dir*, containing the look-up tables and a test product for a random text vector as input.

```
clear all

%%%% Output directory %%%%
dir = 'C:\matrix_vector_mult\preprocessed_files\';
%%%

%%% PARAMETERS %%%
matrix_size = 512;
epsilon = 4/9;
```

```
folding_factor = 2;
%%%%

num_nodes = matrix_size/(folding_factor*data_size);
lookup_size = 2^(epsilon*log2(matrix_size));
data_size = epsilon*log2(matrix_size);
.
.
%%% Generate LUTs %%%
for i=1:folding_factor*num_nodes
    m=mod(i-1,num_nodes)+1;
    for k=0:(lookup_size-1)
        edges = [];
        for j=1:folding_factor*num_nodes
            edges = [edges mod(A(data_size*(j-1)+1:data_size*(j
   -1)+data_size,data_size*(i-1)+1:data_size*(i-1)+data_size)*
   dec2bin(k,data_size)',2)'*exp_mat'];
        end
        RAM_content = dec2bin(reshape(edges',[(size(edges,1)*
   size(edges,2)) 1]),data_size);
        dlmwrite (strcat(preprocessed_data_file,num2str(m),'.
   data'),RAM_content,'-append','delimiter','');
        dlmwrite (strcat(preprocessed_text_file,num2str(m),'.
   txt'),RAM_content,'-append','delimiter',' ');
    end
end
.
.
```

Listing 8.1: pre-process

8.2 NoC partitioning

Parameters:

- *routers*: List containing all router id's in the NoC to be partitioned.
- *part*: List containing the router id's in the partition.
- *part_no*: Partition number.

- *ports_per_router*: Number of ports in the router module.
- *flit_data_width*: Flit data width in the CONNECT generated NoC (can also be looked up in *connect_paramters.v*).
- *vc_bits*: Number of bits for virtual channels (Should be 1 for non-VC networks. Can be looked up in *connect_paramters.v*).
- *dest_bits*: Bits required for specifying port address (logarithm of number of send/receive ports in the Network).
- *hex_filename*: Prefix of the .hex files generated for routing tables by the CONNECT tool.

Usage: Python script. Place the input NoC file *mkNetworkSimple.v* in the same directory. Command: *python mkNetworkScript.py*
Output: Output partitioned network file *mkNetworkSimple_(part_no).v*.

```
import re

########## PARAMETERS ###############
routers =  [0,1,2,3]
part = [0,1,2]
part_no = 0
ports_per_router = 3
flit_data_width = 16
vc_bits = 1
dest_bits = 2
data_width = 2
#hex_filename = "
   mesh_4RTs_2VCs_8BD_16DW_SepIFRoundRobinAlloc_2RTsPerRow_2RTsPerCol_routing
   "
hex_filename = "
   double_ring_4RTs_2VCs_8BD_16DW_SepIFRoundRobinAlloc_routing"
#ALSO CHECK UART BAUD RATE/FREQUENCY#
#####################################
.
.
.
main()
```

Listing 8.2: NoC partitioning script mkNetworkScript.py

8.3 Harware BMVM

Using the hardware BMVM requires the following steps (in order) after the RIFFA driver is correctly set up:

1. Set the parameters in the BMVM parameter file *matrix_vector_mult_paramters.v*.
2. Appropriately instantiate and connect the (partitioned) NoC in the top module *mult_top.v*.
3. Simulate and obtain the *max_cycles* for a single iteration, and set the parameter value in the file *multNode.v*.
4. Generate the bitfile and configure the FPGA (script available in project folder).
5. Reboot the CPU with which FPGA is connected.
6. Specify the number of iterations and the input vector in the c++ file *bmvm.c*.
7. Run *make*.
8. Execute the generated executable using the command *bmvm 2 <fpga_id> 0 <num_iter>*. The <fpga_id> is 0x6018 for our setup.

```
`define RAM_WIDTH 8
`define RAM_ADDR_BITS 12
`define VECTOR_SIZE 64
`define NUM_WORDS 2
`define VERTEX_WIDTH 8
`define NUM_NODES 4
`define ADDR_WIDTH 2
`define FOLDING_FACTOR 2
`define NUM_PARTS 8
`define FOLDING_BITS 1
```

Listing 8.3: BMVM paramter file matrix_vector_mult.v

```
`include "connect_parameters.v"
`include "matrix_vector_mult_parameters.v"


module mult_top(
```

```
    reset,
    clk,
    data_in,
    data_o,
    done
  );

.
.
  wire ser_in_3_3,  in_ports_3_3_getNonFullVCs, ser_in_3_4,
    in_ports_3_4_getNonFullVCs, ser_out_3_2,
    out_ports_3_2_getNonFullVCs, ser_out_3_1,
    out_ports_3_1_getNonFullVCs;
  wire ser_in_1_2, in_ports_1_2_getNonFullVCs, ser_in_2_1,
    in_ports_2_1_getNonFullVCs, ser_out_2_3,
    out_ports_2_3_getNonFullVCs, ser_out_1_4,
    out_ports_1_4_getNonFullVCs;

  assign ser_in_3_3 = ser_out_2_3;
  assign ser_in_3_4 = ser_out_1_4;
  assign out_ports_2_3_getNonFullVCs =
    in_ports_3_3_getNonFullVCs;
  assign out_ports_1_4_getNonFullVCs =
    in_ports_3_4_getNonFullVCs;

  assign ser_in_1_2 = ser_out_3_2;
  assign ser_in_2_1 = ser_out_3_1;
  assign out_ports_3_2_getNonFullVCs =
    in_ports_1_2_getNonFullVCs;
  assign out_ports_3_1_getNonFullVCs =
    in_ports_2_1_getNonFullVCs;


genvar i;
generate
for (i=0; i<folding_factor; i=i+1)
  begin:m
  assign data_o[(folding_factor-i)*vertex_width*4-1:(
    folding_factor-i-1)*vertex_width*4] = {vertex_0[(i+1)*
    vertex_width-1:i*vertex_width], vertex_1[(i+1)*vertex_width
    -1:i*vertex_width], vertex_2[(i+1)*vertex_width-1:i*
```

```
    vertex_width], vertex_3[(i+1)*vertex_width-1:i*vertex_width
    ]};

  assign {getVertex_0[(i+1)*vertex_width-1:i*vertex_width],
    getVertex_1[(i+1)*vertex_width-1:i*vertex_width],
    getVertex_2[(i+1)*vertex_width-1:i*vertex_width],
    getVertex_3[(i+1)*vertex_width-1:i*vertex_width]} = data_in
    [(folding_factor-i)*vertex_width*4-1:(folding_factor-i-1)*
    vertex_width*4];
    end
endgenerate



  mkNetworkSimple_0 dut
  (.CLK(clk)
   ,.RST_N(rst_n)
  , .send_ports_0_putFlit_flit_in(send_ports_0_putFlit_flit_in)
  , .EN_send_ports_0_putFlit(EN_send_ports_0_putFlit)
  , .EN_send_ports_0_getNonFullVCs(1'b1)
  , .send_ports_0_getNonFullVCs(send_ports_0_getNonFullVCs)
  , .recv_ports_0_getFlit(recv_ports_0_getFlit)
  , .EN_recv_ports_0_getFlit(EN_recv_ports_0_getFlit)
  , .EN_recv_ports_0_putNonFullVCs(1'b1)
  , .recv_ports_0_putNonFullVCs_nonFullVCs('b1)
  , .recv_ports_info_0_getRecvPortID(
   recv_ports_info_0_getRecvPortID)


  , .send_ports_1_putFlit_flit_in(send_ports_1_putFlit_flit_in)
  , .EN_send_ports_1_putFlit(EN_send_ports_1_putFlit)
  , .EN_send_ports_1_getNonFullVCs(1'b1)
  , .send_ports_1_getNonFullVCs(send_ports_1_getNonFullVCs)
  , .recv_ports_1_getFlit(recv_ports_1_getFlit)
  , .EN_recv_ports_1_getFlit(EN_recv_ports_1_getFlit)
  , .EN_recv_ports_1_putNonFullVCs(1'b1)
  , .recv_ports_1_putNonFullVCs_nonFullVCs('b1)
  , .recv_ports_info_1_getRecvPortID(
   recv_ports_info_1_getRecvPortID)
```

```
, .send_ports_2_putFlit_flit_in(send_ports_2_putFlit_flit_in)
, .EN_send_ports_2_putFlit(EN_send_ports_2_putFlit)
, .EN_send_ports_2_getNonFullVCs(1'b1)
, .send_ports_2_getNonFullVCs(send_ports_2_getNonFullVCs)
, .recv_ports_2_getFlit(recv_ports_2_getFlit)
, .EN_recv_ports_2_getFlit(EN_recv_ports_2_getFlit)
, .EN_recv_ports_2_putNonFullVCs(1'b1)
, .recv_ports_2_putNonFullVCs_nonFullVCs('b1)
, .recv_ports_info_2_getRecvPortID(
 recv_ports_info_2_getRecvPortID)

,.ser_in_1_2(ser_in_1_2)
,.ser_out_1_4(ser_out_1_4)
,.ser_in_2_1(ser_in_2_1)
,.ser_out_2_3(ser_out_2_3)
,.out_ports_1_4_getNonFullVCs(out_ports_1_4_getNonFullVCs)
,.in_ports_1_2_getNonFullVCs(in_ports_1_2_getNonFullVCs)
,.out_ports_2_3_getNonFullVCs(out_ports_2_3_getNonFullVCs)
,.in_ports_2_1_getNonFullVCs(in_ports_2_1_getNonFullVCs)

 );

mkNetworkSimple_1 dut1
(.CLK(clk)
 ,.RST_N(rst_n)

, .send_ports_3_putFlit_flit_in(send_ports_3_putFlit_flit_in)
, .EN_send_ports_3_putFlit(EN_send_ports_3_putFlit)
, .EN_send_ports_3_getNonFullVCs(1'b1)
, .send_ports_3_getNonFullVCs(send_ports_3_getNonFullVCs)
, .recv_ports_3_getFlit(recv_ports_3_getFlit)
, .EN_recv_ports_3_getFlit(EN_recv_ports_3_getFlit)
, .EN_recv_ports_3_putNonFullVCs(1'b1)
, .recv_ports_3_putNonFullVCs_nonFullVCs('b1)

,.ser_in_3_3(ser_in_3_3)
,.ser_out_3_1(ser_out_3_1)
,.ser_in_3_4(ser_in_3_4)
,.ser_out_3_2(ser_out_3_2)
,.out_ports_3_1_getNonFullVCs(out_ports_3_1_getNonFullVCs)
,.in_ports_3_3_getNonFullVCs(in_ports_3_3_getNonFullVCs)
```

```
  ,.out_ports_3_2_getNonFullVCs(out_ports_3_2_getNonFullVCs)
  ,.in_ports_3_4_getNonFullVCs(in_ports_3_4_getNonFullVCs)
  , .recv_ports_info_3_getRecvPortID(
   recv_ports_info_3_getRecvPortID)
  );

.
.

  multNode node_0 (
        .reset(reset),
        .clk(clk),
        .getID(recv_ports_info_0_getRecvPortID),
        .getVertex(getVertex_0),
        .flit_in(recv_ports_0_getFlit),
        .rcvReady(EN_recv_ports_0_getFlit),
        .canSend(send_ports_0_getNonFullVCs[0]),
        .flit_out(send_ports_0_putFlit_flit_in),
        .putFlit(EN_send_ports_0_putFlit),
        .LUT_address(LUT_address_0),
        .LUT_output_data(LUT_output_data_0),
        .vertex(vertex_0),
        .done(done)
    );
.
.

endmodule
```

Listing 8.4: Top module mult_top.v

```
module multNode(reset, clk, getID, getVertex, flit_in, getFlit,
    rcvReady, canSend, flit_out, putFlit,
        LUT_address, LUT_output_data, vertex, done);
.
.


  always @ (posedge reset, posedge clk)
  begin
```

```
    if(reset)
  begin
    myID <= getID;
    rcvReady <= 0;
    flit_out <= 0;
    state <= 0;
    vc <= 0;
    send_count <= 0;
    recv_count <= 0;
    LUT_address <= (getVertex[vertex_width-1:0] << (addr_width
   + folding_bits)) + getID;
    temp_vertex <= getVertex;
    vertex <= 0;
    cycle <= 0;
    send_count <= 0;
    dest <= getID;
    flag <= 1;
    done <= 0;
    iter <= 0;
    display_iter_time <= 1;
end

else
  begin
   cycle <= cycle + 1;
   if (flit_in[flit_port_width-1]) // valid flit
   begin
       vertex <= next_vertex;
       recv_count <= recv_count + 1;
   end

   if(state == 0)
   begin
     if ((!flit_in[flit_port_width-1]) && (flag == 0))
     begin
       LUT_address[addr_width + folding_bits - 1: 0] <=
  LUT_address[addr_width + folding_bits - 1: 0] + 1;
       vertex <= init_vertex;
       dest <= (dest + 1);// % num_nodes;
       rcvReady <= 1;
       state <= 1;
```

```
      send_count <= send_count + 1;
      flag <= 1;
    end
    else if (flag == 1)
      flag <= 0;
 end

 else if(state == 1)
 begin
    if ((putFlit == 1) && (flag == 0))// && (myID == 63))
    begin
      flag <= 1;
      LUT_address[addr_width + folding_bits - 1: 0] <=
LUT_address[addr_width + folding_bits - 1: 0] + 1;
      dest <= (dest + 1);// % num_nodes;
      send_count <= send_count + 1;
      flit_out <= {1'b1 /*valid*/, 1'b1 /*tail*/, dest[
addr_width-1:0], vc, myID, part_no, LUT_output_data};

      if (send_count[folding_bits+addr_width-1:0] == (
num_parts - 1))
      begin
        state <= 3;
        temp_vertex <= {temp_vertex[vertex_width-1:0],
temp_vertex[(folding_factor*vertex_width)-1:vertex_width]};
        rcvReady <= 0;
       end
      else if(send_count[addr_width-1:0] == (num_nodes - 1))
      begin
        state <= 2;
        flag <= 1;
        rcvReady <= 0;
      end
      else
      begin
        rcvReady <= 1;
      end
    end
    else if (flag == 1)
    begin
```

```
      if (send_count < recv_count + `FLIT_BUFFER_DEPTH + (
iter << folding_bits))
        flag <= 0;
    end
  end
  if(state == 2)
  begin
    if ((!flit_in[flit_port_width-1]) && (flag  == 0))
    begin
      state <= 1;
      send_count <= send_count + 1;
      vertex <= init_vertex;
      LUT_address[addr_width + folding_bits - 1: 0] <=
LUT_address[addr_width + folding_bits - 1: 0] + 1;
      dest <= dest + 1;
      flag <= 1;
    end
    else if (flag  == 1)
    begin
      flag <= 0;
    end

  end
  if(state == 3)
  begin
    if (send_count[2*folding_bits+addr_width:folding_bits+
addr_width] < folding_factor)
    begin
      LUT_address <= (temp_vertex[vertex_width-1:0] << (
addr_width + folding_bits)) + getID + (send_count[2*
folding_bits+addr_width-1:folding_bits+addr_width] << (
folding_bits + vertex_width + addr_width));
      state <= 0;
      flag <= 1;
      rcvReady <= 0;
    end
    else
    begin
      state <= 4;
      rcvReady <= 1;
    end
```

```
    end
    if (state == 4)
    begin
      if(putFlit == 1)
      begin
        state <= 5;
        iter <= iter + 1;
      end
    end
    if (state == 5)
    begin
      if ((cycle > max_cycle) && (iter < max_iter))
      begin
         state <= 0;
           rcvReady <= 0;
         flit_out <= 0;
         vc <= 0;
         send_count <= 0;
         recv_count <= 0;
         LUT_address <= (vertex[vertex_width-1:0] << (
   addr_width + folding_bits)) + myID;
         temp_vertex <= vertex;
         vertex <= 0;
         cycle <= 0;
         send_count <= 0;
         dest <= myID;
      end
      else if((cycle > max_cycle) && (iter >= max_iter))
      begin
        done <= 1;
      end

    end
  end
end

.
.
endmodule
```

Listing 8.5: Processing Element multNode.v

8.4 Software BMVM

Parameters:

- *size*: Column size (n) of the matrix A.
- *folding*: This parameter should be set to the folding factor f.
- *num_threads*: Should be equal to n/k.
- *num_iter*: Number of iterations in matrix vector multiplication. Setting it to value i would produce $A^i v$ as output.
- *lookup_size*: Set this value to 2^k.

Usage: Compile using *g++ -o out multiply.cpp -lpthread* . Run using *./out*.
Output: Displays the output vector $A^i v$, where i is the number of iterations.

```
.
.
#define size 1024
#define num_threads 64
#define lookup_size 16
#define folding 4
#define num_iter 1000

#define data_size size/(num_threads*folding)

pthread_mutex_t locks[num_threads];
pthread_barrier_t barrier;

.
.

struct thread_data
{
  int thread_id;
  int lookup[folding][lookup_size][folding][num_threads][
   data_size];
  int v[folding];
  int iter;
  thread_data()
```

```
  {
  for (int f = 0; f < folding; f++)
    v[f] = 0;
  iter = 0;
  }
};

void *multiply(void* input)
{
  thread_data *data;
  data  = (struct thread_data *) input;
  int loop = 0;
  int repeat = 1;
  int i;
  while (repeat > 0)
  {
    for(int f1 = 0; f1 < folding; f1++)
    {
      int v = data->v[f1];
      for (int f2 = 0; f2 < folding; f2++)
      {
        for (int l = 0; l < num_threads; l++)
        {
          //i = (l + data->thread_id) % num_threads;
          pthread_mutex_lock(&(locks[l]));
          for (int m = 0; m < data_size; m++)
          {
            v_bar[f2][l][m] = bitXor(v_bar[f2][l][m], data->
   lookup[f1][v][f2][l][m]);
          }
          pthread_mutex_unlock(&(locks[l]));
        }
      }
    }
    loop++;
    if (loop == num_iter)
      repeat = 0;
    else
    { pthread_barrier_wait(&barrier);

      pthread_mutex_lock(&(locks[data->thread_id]));
```

```
      for (int f = 0; f < folding; f++)
      {
        data->v[f] = 0;
        for (int m = 0; m < data_size; m++)
        {
          data->v[f] += v_bar[f][data->thread_id][m] * pow(2,
   data_size - m -1) ;
          v_bar[f][data->thread_id][m] = 0;
        }
      }
      pthread_mutex_unlock(&(locks[data->thread_id]));
      pthread_barrier_wait(&barrier);
    }

  }

  pthread_exit(NULL);
}

.
.

int main()
{
    pthread_t threads[num_threads];
    pthread_barrier_init(&barrier, NULL, num_threads);

    thread_data *input;//[num_threads];
    input = new thread_data[num_threads];

.
.

    for (int i = 0; i<num_threads; i++)
    {
        pthread_create(&threads[i], NULL, multiply, (void*) &
   input[i]);
    }

    for (int i = 0; i<num_threads; i++)
    {
```

```
        pthread_join(threads[i], NULL);
    }
.
.
  return 0;
}
```

Listing 8.6: Software BMVM multiply.cpp

References

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

[2] 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*. IEEE, 2004, pp. 263–270.

[3] L. Barnault and D. Declercq, "Fast decoding algorithm for ldpc over gf(2q)," in *Information Theory Workshop, 2003. Proceedings. 2003 IEEE*, March 2003, pp. 70–73.

[4] D. J. Bernstein, "Circuits for integer factorization: a proposal," *At the time of writing available electronically at http://cr. yp. to/papers/nfscircuit. pdf*, 2001.

[5] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny, "The power of priority: Noc based distributed cache coherency," in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*. IEEE, 2007, pp. 117–126.

[6] Ü. V. Çatalyürek and C. Aykanat, "Decomposing irregularly sparse matrices for parallel matrix-vector multiplication," in *Parallel Algorithms for Irregularly Structured Problems*. Springer, 1996, pp. 75–86.

[7] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 2, p. 15, 2009.

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

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

[10] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*. IEEE, 2001, pp. 684–689.

[11] J.-P. Diguet, S. Evain, R. Vaslin, G. Gogniat, and E. Juin, "Noc-centric security of reconfigurable soc," in *Proceedings of the First International Symposium on Networks-on-Chip*. IEEE Computer Society, 2007, pp. 223–232.

[12] S. Fuller, *RapidIO: The embedded system interconnect*. John Wiley & Sons, 2005.

[13] R. L. Geiger, P. E. Allen, and N. R. Strader, *VLSI design techniques for analog and digital circuits*. McGraw-Hill New York, 1990, vol. 90.

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

[15] M. Gschwind, V. Salapura, and D. Maurer, "Fpga prototyping of a risc processor core for embedded applications," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 2, pp. 241–250, 2001.

[16] 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*. IEEE, 2013, pp. 1–8.

[17] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian, "Compressing large boolean matrices using reordering techniques," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 13–23.

[18] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*. IEEE, 2002, pp. 105–112.

[19] 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*. ACM, 1990, pp. 564–572.

[20] 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.

[21] 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*. IEEE, 2007, pp. 110–115.

[22] 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*. Springer, 1998, pp. 31–36.

[23] 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*. ACM, 2012, pp. 37–46.

[24] L.-S. Peh, S. W. Keckler, and S. Vangal, "On-chip networks for multicore systems," in *Multicore Processors and Systems*. Springer, 2009, pp. 35–71.

[25] 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*. ACM, 2003, pp. 100–107.

[26] 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*. IEEE, 1995, pp. 146–152.

[27] P. Specification, "Virtex-5 family overview," 2006.

[28] M. D. Swanson and A. H. Tewfik, "A binary wavelet decomposition of binary images," *Image Processing, IEEE Transactions on*, vol. 5, no. 12, pp. 1637–1650, 1996.

[29] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," *IBM Journal of research and development*, vol. 41, no. 6, pp. 711–725, 1997.

[30] C. A. Valderrama, A. Changuel, and A. A. Jerraya, "Virtual prototyping for modular and flexible hardware-software systems," *Design Automation for Embedded Systems*, vol. 2, no. 3-4, pp. 267–282, 1997.

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

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

[33] R. E. Wunderlich and J. C. Hoe, "In-system fpga prototyping of an itanium microarchitecture," in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*. IEEE, 2004, pp. 288–294.