NANYANG TECHNOLOGICAL UNIVERSITY

EVALUATING THE PERFORMANCE OF COMPUTING PLATFORMS USING A SET OF COMPUTE KERNELS

by

ADHIKARI SAURABH
(G1601326D)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2017

Contents

List of Figures

List of Tables

Abbreviations

ACP Accelerator Coherency Port

API Asymmetric Multiprocessing

BRAM Block Random Access Memory

BCSR Blocked Compressed Sparse Row

CSR Compressed Sparse Row

CSC Compressed Sparse Column

DFG Data Flow Graph

DMA Direct Memory Access

FPGA Field Programmable Gate Arrays

FSBL First Stage Boot Loader

MXP Matrix Processor

PL Programmable Logic

PS Processing system

SpMV Sparse Matrix Vector

Abstract

Research efforts have shown strength of hardware accelerators in a wide range of application domains where compute kernels can execute efficiently on an accelerator. One such example is NEON hard vector engine coupled with ARMv7 32-bit processor on Xilinx Zynq Platform where both of these can run at 667 MHz. For byte-level operations, ARMv7 processor is able to provide a peak performance of 0.66 Giga-operations per second (GOPS). Despite the fact that NEON can perform 8 byte-level operations every clock cycle and able to provide a peak performance of 5.3 GOPS, most compute kernels suffer to take advantage of the performance of NEON engine. For example, ARMv7 and NEON can provide a performance of only 325 MOPS and 990 MOPS, respectively for quadratic polynomial (from [1]). On the other hand, 64-lane MXP soft processor (a carefully designed vector-engine) running at 110 MHz can provide a peak performance of 7 GOPS and while executing quadratic polynomial it can provide a performance of 1.51 GOPS, which is much faster than ARMv7 (4.75×) and NEON (1.56×).

In this report, we first analyze compute kernels by extracting data flow graphs and then evaluate the performance of computing platforms such as ARMv7, NEON, Intel processor, and MXP soft vector-engine. The goal is to quatify the gap between peak performance of the platform and achievable performance for a set of compute kernels. Major focus of this report is on MXP soft vector-engine which we instantiate on Zynq device available on Zedboard platform. Apart from using compute kernels to evaluate the performance, we also use an image processing application and a complex benchmark (SpMV) to better understand the performance gap.

Acknowledgment

I would like to express my deep and sincere gratitude to Associate Professor Dr. Douglas Leslie Maskell, for his constant and continuous support, encouragement and guidance. I would also like to acknowledge the crucial role of mentor Dr. Abhishek Kumar Jain, for his continuous support, effective suggestions and professional guidance in the entire phase of my dissertation. I sincerely thank him for arranging weekly meeting which were helpful in discussing the project to find the right path to proceed. I would also like to thank my friends and classmates for helping me with technical issues regarding my practical work. My heartfelt appreciation goes to my beloved parents, Mr. Harsh Singh and Mrs. Bimla, for their support and love throughout my studies at the University. Finally, I would like to thank School of Computer Engineering, Nanyang Technological University, Singapore for their support.

Chapter 1

Introduction

1.1 Motivation

In a typical signal processing application, 20% of the program code consumes 80% of the application execution time. This short section of code generally contains compute intensive arithmetic operations which we refer to as compute kernel. A General Purpose Processor (GPP) can be used for the execution of compute kernels by describing their functionality using C or C like programming languages. With the advancements in technology, parallel processing architectures such as multi-cores CPUs and DSPs, GPUs, Massively parallel processor arrays, FPGA based accelerators are gaining popularity for accelerated execution of kernels. Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge that conventional processor designs will not be able to meet.

Hardware accelerators, such as vector-engines and graphics processing units have been shown to be effective when paired with general purpose processors, offering software-like programmability and improved performance. One such example is NEON hard vector engine coupled with ARMv7 32-bit processor on Xilinx Zynq Platform where both of these can run at 667 MHz as shown in Figure 1.1. It is possible to offload the execution of compute kernels from ARMv7 processor to NEON hard vector engine for efficient processing of data-parallel kernels.

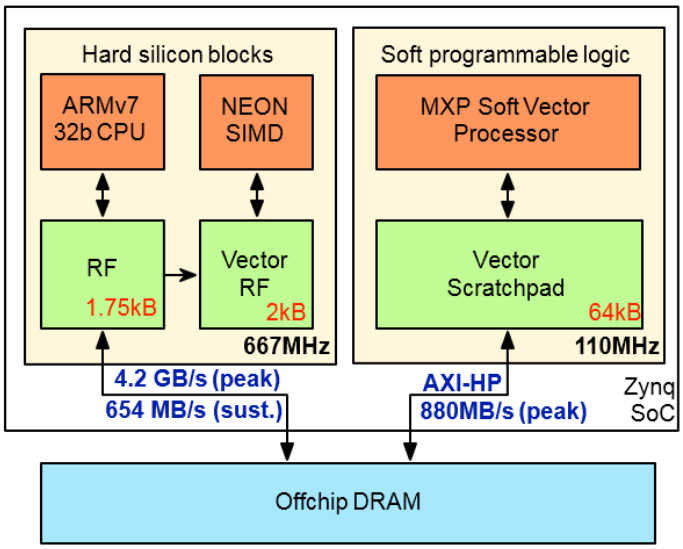


Figure 1.1: Xilinx Zynq SoC Platform Compute Organization [1]

Metric	ARMv7 CPU	NEON	MXP	Ratio
μarch	Scalar	SIMD	Soft vector	
Clock Freq.	667 MHz	667 MHz	110 MHz	1/6×
Lanes	1	2×32b	1–16×32b	8×
		4×16b	2–32×16b	8×
		8×8b	4–64×8b	8×
Throughput				
32b (Gops/s)	0.6	1.3	1.7	1.3×
16b (Gops/s)	0.6	2.6	3.5	1.3×
8b (Gops/s)	0.6	5.3	7	1.3×
Memory	1.75kB (Scalar RF)	2kB (Vector RF)	4–256kB (Scratchpad)	2–128×

Figure 1.2: Architecture Specifications [1]

As shown in Figure 1.2, For byte-level operations, ARMv7 processor is able to provide a peak performance of 0.66 Giga-operations per second (GOPS). Despite the fact that NEON can perform 8 byte-level operations every clock cycle and able to provide a peak performance of 5.3 GOPS, most compute kernels suffer to take advantage of the performance of NEON engine.

For example, ARMv7 and NEON can provide a performance of only 325 MOPS

```
// simple for loop
for(int i=0;i<N;i++) {
        // computing ax^2 + bx + c
        y[i]=a  *(x[i])*(x[i])+b  *x[i]+c;
}
```

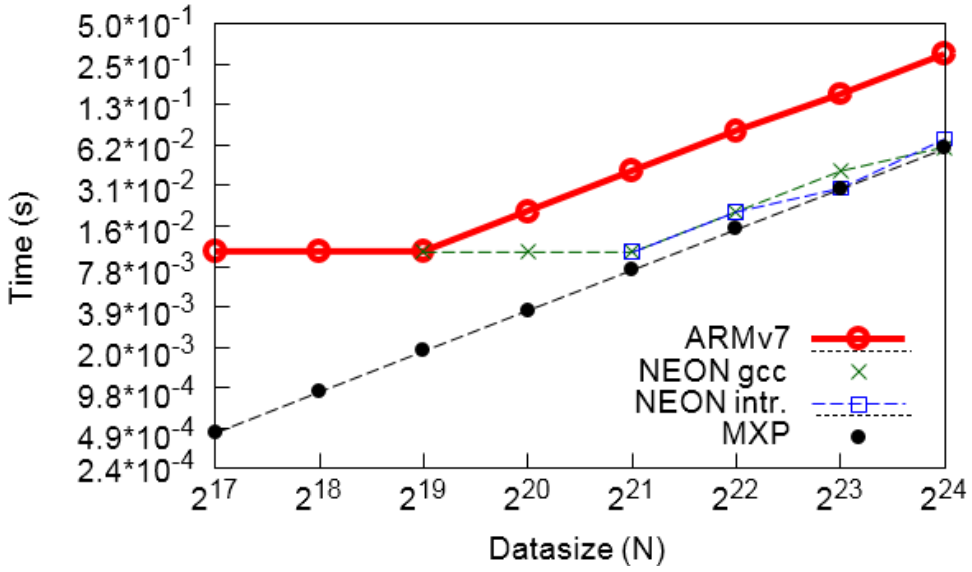


Figure 1.3: Comparing runtime for $a * x^2 + b * x + c$ [1]

and 990 MOPS, respectively for quadratic polynomial (as shown in Figure 1.3). On the other hand, 64-lane MXP soft processor (a carefully designed vector-engine) running at 110 MHz can provide a peak performance of 7 GOPS and while executing quadratic polynomial it can provide a performance of 1.51 GOPS, which is much faster than ARMv7 (4.75×) and NEON (1.56×).

In this report, we first analyze compute kernels by extracting data flow graphs and then evaluate the performance of computing platforms such as ARMv7, NEON, Intel processor, and MXP soft vector-engine. The goal is to quatify the gap between peak performance of the platform and achievable performance for a set of compute kernels. Major focus of this report is on MXP soft vector-engine which we instantiate on Zynq device available on Zedboard platform. Apart from using compute kernels to evaluate the performance, we also use a simple image processing application and a complex benchmark (SpMV) to better understand the performance gap.

1.2 Contribution

The main contributions can be summarized as follows:

- Setting up Linux and accessing MXP overlay through it so that in future much more attention can be given on building up the MXP application rather than struggling to configure Linux for the MXP overlay.
- Comparing the MXP overlay performance against other processors like Intel I3, ARMv7 and NEON hard vector engine for a set of compute kernels.
- Accelerating image processing application and a complex benchmark (SpMV) using MXP APIs.

1.3 Organization

The remainder of the report is organized as follows:

Chapter 2 gives background information about different computing platforms and most importantly about the MXP vector processor, refer to as an overlay as well. In chapter 3, we describe about the MXP vector processor, it's architecture in detail and the concept of overlapping computation with communication that make it more effective in terms of throughput as compared to other embedded hard vector processors. In chapter 4, we discuss how we configure Linux for MXP on Xilinx ZedBoard so that we can provide OS support for MXP on the Xilinx ZedBoard. In chapter 5, we describe about our experimentation for the performance analysis of MXP vector processor while accelerating compute kernels. In chapter 6, we describe about our experimentation for runtime analysis of image processing application and SpMV computational kernel. We conclude in chapter 7 and discuss future work.

Chapter 2

Background

2.1 High Performance Computing

Accelerated Computing also known as computational acceleration has led to an era of high performance computing (a.k.a HPC). Moore's law has been widely accepted as an industry standard and these industries were highly dependent on Moore's law for better performance and improved efficiency. It was cited by most of the semiconductor manufacturers as they strive to increase computational power. Every year, the transistors that were being added on the silicon area were very fast and consumed less amount of power. But, in recent years, Moore's Law has slowed down by a considerable amount since as the number of the transistor on the chip increases, the frequency stopped scaling. Hence, it became very difficult to extract the performance from a single core processor (sequential CPU). Moreover, single processor ran into memory wall and power wall issues. Thus, adding more CPU cores was a solution from where the multicore processor emerged. However, with multiple CPU cores, it became difficult to gain better performance out of these chips. The challenge was writing the code that can run across multiple cores simultaneously. Moreover, some instructions can't run simultaneously at all. So, the computing world ended up with multicore CPUs that could not accelerate all types of code. This made the designers job even harder. Simply adding cores resulted into waste of transistors and raised the cost to manufacture the processor without much benefit. Failure to improve the

performance of CPU, without affecting power budgets or using extremely complicated design methods resulted in the industry hitting a brick wall.

Accelerated Computing has reached a tipping point in the High-Performance Computing. Within a year or two, the majority of the system will be equipped with accelerators. [2]

Heterogenous computing refers to systems that make use of more than one kind of processors. Performance gain or energy efficiency is achieved in these systems by adding dissimilar coprocessors for handling multiple type of tasks using their specialized computational/processing capabilities. Accelerated computing is a computing model used for accelerating applications in engineering and scientific domains wherein the computations are performed on specialized processors accelerators. It makes use of heterogeneous computing systems for improving the performance of application in which a lot of data is executed in parallel. The basic idea is to execute the code that is suitable for a processor. For example, sequential code with a lot of control instructions and branches would be preferred to run over a CPU since they would provide better performance for this kind of code, whereas computation which is highly parallel with very less branching conditions would be well suited for execution on the accelerator.

2.2 FPGA Virtualization for High Performance Computing

Virtualization of FPGA using the FPGA overlays delivers high performance for application acceleration [3]. With the increasing complexity of FPGA platforms, it is being said that the use of overlay architecture will become mainstream. Overlay architecture can enable widespread use of the FPGAs in accelerated computing. FPGAs make the full utilization and advantage of Moore's Law improvements in semiconductor technology [4]. Reconfigurable platforms consisting of general-purpose processors along with the programmable logic have been introduced by the major FPGA vendors like Xilinx and Altera. FPGAs are used for high speed computations for the data

parallel applications. However, it remains difficult to develop an accelerator using Hardware Description Language (HDL). It requires expertise in hardware designing performing implementation and debugging for building hardware. Hardware design, design productivity and long compilation times are few of the barriers that restricts the use of FPGA in general purpose computing.

FPGA virtualization using Overlay architecture offers the advantage of the fast compilation, run-time management and software-like programmability because of their improved design productivity and high-level design abstraction [5], [6],[7]. Other benefits include better design reuse, application portability across platforms and rapid reconfiguration that is much faster than partial reconfiguration on fine grained FPGAs. Integrating overlays with memory subsystem and processor is important to enable sharing and management of limited overlay resources. Overlays exhibit features independent from the host FPGA. Soft processors can extend the capability of embedded hard vector processors in FPGA like Xilinx Zynq.

In this thesis work, we are going to use an overlay architecture known as MXP soft-processor used for high performance computing in our experiments while analysing the runtime, speedup and throughput obtained. It is a soft vector processor developed by Vectorblox Computing Inc [8] and is classified as Single Instruction, Multiple Data (SIMD) Stream processor. It is provided as an IP core which can be instantiated over the FPGA. It provides the acceleration of the data parallel operations. MXP Vectorblox can be programmed in C/C++ and provides different C/C++ APIs support. This helps to program it easily rather than struggling a lot with Hardware Description Languages like VHDL or Verilog. For using an accelerator over FPGA, the hardware designing flow requires to go through a long design cycle (taking up hours to weeks) whereas on the other hand using MXP, all we need to do is to alter the given software code. MXPs parameterized design helps user to specify the amount of parallelism needed, which can range from 1 to 128. It includes a parallel access local scratchpad memory which holds the data and a high-throughput Direct Memory Access (DMA). MXP can be easily instantiated in the existing Xilinx and Altera development boards, simplifying the development.

In our work we have used Xilinx Zynq 7020 SoC [9] heterogeneous computing

platform wherein the MXP soft-processor is instantiated on its programmable logic fabric. Now, we will discuss about the hardware platform which we have used in detail.

2.3 Zynq-7000 SOC Board

Xilinx ZedBoard is a SOC development board which is based on Xilinx Zynq-7000 All Programmable SoC (AP SoC) [10]. ZedBoard is a platform which is partitioned into Processing system (PS) consisting of one or multiple processors along with memory interfaces, bus and peripherals and the Programmable Logic. It provides a way in which even a customized hardware can be instantiated. These two parts are connected via high throughput interconnect to maximize communication bandwidth. ZedBoard features are shown below in table 2.1.

Feature	**Description**
Processor	Zynq-7000 AP SoC XC7Z020-CLG-484-1
Memory	512 MB DDR3 256 MB Quad-SPI Flash 4.0 GB SD card
Communication	Onboard USB-JTAG Programming USB OTG 2.0 and USB-UART 10/100/1000 Ethernet
Expansion Connectors	Expansion connectors FMC-LPC connector Five Pmod compatible header(2x6) Agile Mixed Signalling (AMS) header
Clocking	33.33 MHz clock source for PS 100 Mhz oscillator for PL
Display	HDMI output supporting 1080p60 with 16-bit YCbCr 4:2:2 mode color VGA output (12-bit resolution color)
Configuration and Debug	Onboard USB-JTAG interface Xilinx Platform Cable JTAG connector
General Purpose I/O	Eight user LEDs Seven push buttons Eight DIP switches

Table 2.1: ZedBoard Features

Figure 2.1 represents ZedBoard block diagram [11]. It gives a brief idea about the Zynq xc7z020-clg484 System On Chip Development Board.

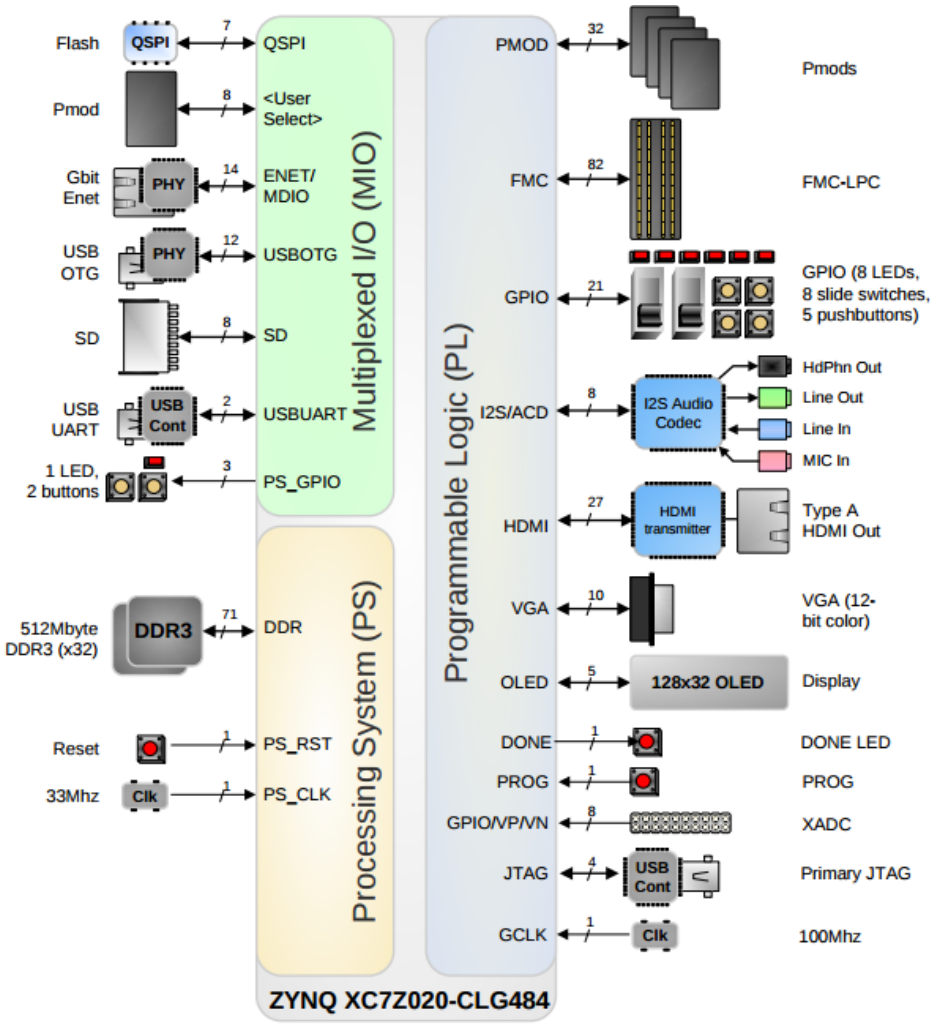


Figure 2.1: ZedBoard Block Diagram[11]

FPGA fabric provides the power of reconfigurability along with application specific acceleration while letting the processor execute control intensive tasks. Xilinx Zynq ZedBoard is a development and evaluation board which is based on Zynq-7000 All Programmable SoC architecture. It consists of Zynq Z7020-clg484 of speed grade -1(667 MHz) containing dual core ARM-Cortex A9 based processing system(PS) and PL fabric in one package [12]. The PS consists of a floating-point unit (double precision), DMA controller, 512 MB DDR RAM, commonly used peripherals and external memory interfaces. The components of PS are listed as below:

- Two ARM Cortex-A9 cores that are run-time configurable as a single processor.
- NEON 128b SIMD coprocessor and VFPv3 per processor.
- 512KB L2 cache that is shared between the processors.
- 32KB instruction and L1 data caches per processor.
- Snoop Control Unit (SCU) and the ACP for cache coherent accesses.
- On-Chip Memory (OCM) with capacity of 256KB.
- DMA controller with four channels for PS and PL.
- DDR controller.

The PL is made up of Artix 7 FPGA fabric [12]. It has 6-input lookup tables (LUTs) and 36kb Block RAMs which can be configured as two 18 Kb blocks. The processor in the system is first booted and PL is configured as a part of boot process or can be configured sometime later in the future. PL can be either reconfigured completely or partially by making use of the partial reconfiguration (PR) feature. The PL configuration data is referred to as bitstream. PL is useful for real-time applications as it has predictable latency. Power can be managed by powering down the PL as it has a different power domain than the PS. The PL has a rich architecture capable of user configuration which are listed below:

- Configurable logic blocks (CLB) with 6-input lookup table (LUT).

- DSP48E1 Slices consisting ALU useful for Digital signal processing.
- 36KB block RAM with capability of dual port.
- Low jitter clock distribution and low skew.
- High performance I/Os that can be configured.

The ARM based reconfigurable system on ZedBoard makes use of multiple AXI interfaces for communication between the PS and the PL. Each interface provides AXI channels, which helps in efficient data transfer and eliminates performance bottlenecks for memory and I/O. There are three types of AXI interfaces to the fabric mentioned below:

- AXI GP - Two 32-bit AXI master and two 32-bit AXI slave General purpose (GP ports).
- AXI HP - Four 32-bit/64-bit configurable, AXI slave High Performance (HP) ports which are buffered along with direct access to DDR and OCM.
- AXI ACP - One 64-bit AXI Accelerator Coherency Port (ACP) slave interface ensuring access to memory is coherent. AXI ACP - One 64-bit AXI Accelerator Coherency Port (ACP) slave interface.

Figure 2.2 represents the Zynq block diagram consisting of the Processing System and Programmable Logic(Fabric).

2.4 OS Support for Programmable Fabric

Efficient runtime scheduling of the software and the hardware tasks can be achieved using the Operating System support on the programmable fabric [13], [14]. It simplifies the programming model. It relieves the user from the responsibility of managing the physical resources like input/output devices, memory and processors. Usage of reconfigurable platforms helps in efficient memory management, coordinating multiple

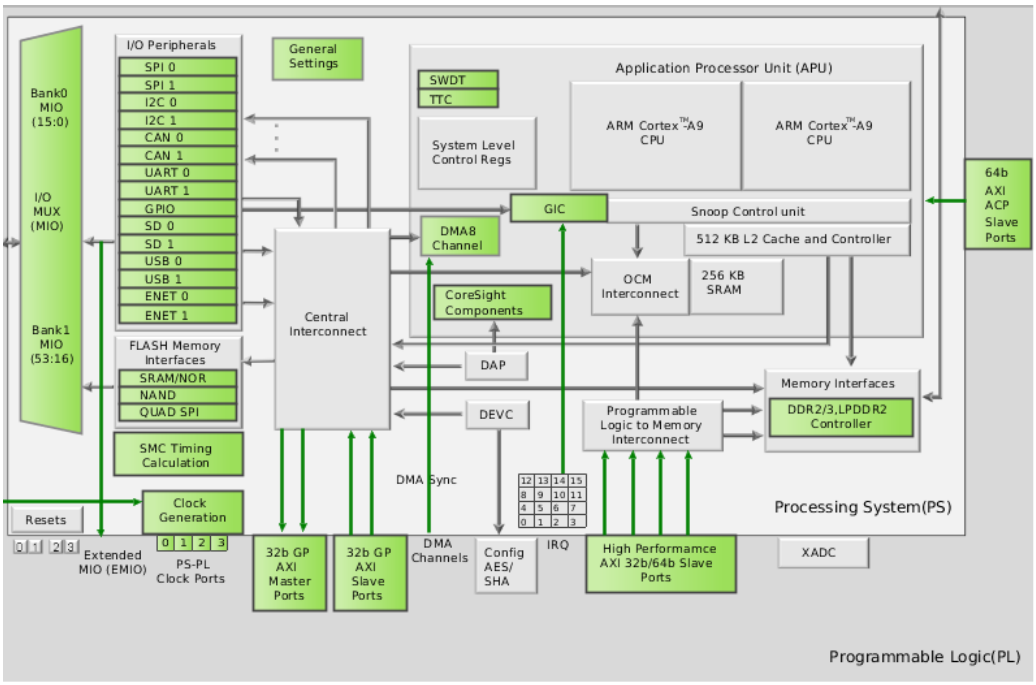


Figure 2.2: Zynq Block Diagram[12]

hardware tasks and managing shared resources across the applications [15]. Particularly, the presence of open-source operating system like Linux provides a good control over scheduling of tasks, synchronization, interrupt management etc. Due to overheads introduced, Linux applications will not perform as efficient compared to bare-metal applications, but as they are heavily abstracted from the underlying hardware, it simplifies application development for software developers. Thus, bare metal application requires explicit handling of the resources, synchronization and communication. Xillybus [16] is among one of them that provides the Operating System support on the FPGA.

We came up with the procedure for setting Linux on ZedBoard for MXP applications. Experience gained from booting Linux on ZedBoard [17] and with some hints from the Linux setup for using MXP [18] made it possible to run Linux on ZedBoard for the MXP applications. Detailed description about the MXP architecture is provided in chapter 3 and the procedure for setting Linux on ZedBoard for the MXP will be discussed in chapter 4. This will help in better understanding of the booting

process and in future we dont have to deal with the set-up issues of Linux for the MXP. Furthermore, in the chapter 5, we will calculate the MXP performance against some standards benchmarks. In the chapter 6, we will have a brief discussion about the performance of MXP while dealing with the image processing application.

Chapter 3

MXP for High Performance Computing

3.1 VectorBlox MXP Matrix Processor

VectorBlox MXP is a scalable supercomputer on FPGA. It is a SIMD accelerator, available as an IP core. MXP consist of the scratchpad memory which is its local memory and all vector operations are performed directly upon it which maximizes its performance. Unlike traditional processors which have address and data registers, there are no load-stores of vector data in MXP [19]. The architecture of MXP is composed of DMA engine and vector engine. MXP soft-processor provides maximum improved performance because the DMA and vector engine can run at a frequency which is different from the host CPU frequency. DMA engine is used to transfer the data to the local scratchpad. Since MXP implements DMA to access the DDR memory directly, the cache-hierarchy is bypassed. Vector engine is composed of multiple parallel vector lanes, the number of which can be configured from 1 to 256 and 4KB of scratchpad is available for each vector lane. Each lane consists of 32-bit ALU and 4KB of scratchpad memory. The scratchpad and ALU can be divided to perform operations on 16-bits/halfword or 8-bits/byte of data. Thus, the architecture of the MXP soft processor during the byte (8-bits) and half-word (16-bits) level operation provides four and two times the performance respectively as compared to the

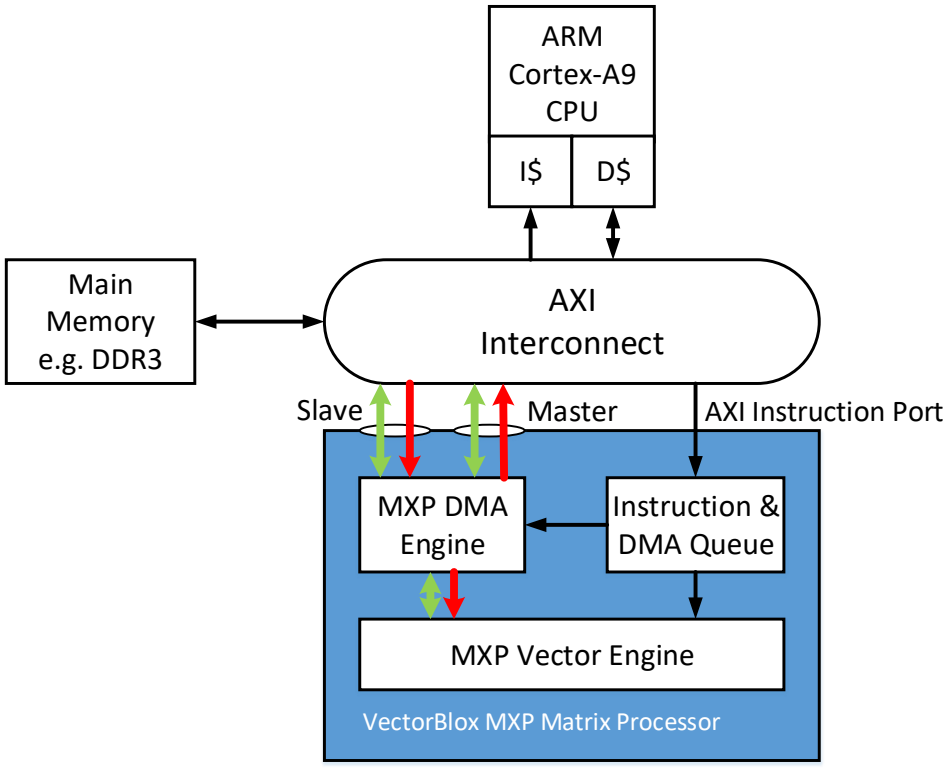


Figure 3.1: Matrix Processor on ZedBoard[19]

performance obtained for word level operation.

MXP vector processor is instantiated onto the programmable logic of the Zynq ZedBoard SoC as shown in Figure 3.1. The communication with ARM processor happens via general purpose ports whereas high performance ports are used for communication with the DDR memory. With 64 KB of scratchpad, we can put 32 16-bit lanes/ 16 32-bit lanes / 64 8-bit lanes. The instruction port uses a dedicated AXI slave interface connecting to master GP port on PS. The scratchpad also connects its slave interface to PS through one of the master GP ports. The DMA engine is connected to the DDR memory controller though AXI slave HP ports. Maximum frequency achievable for the 16-lane MXP soft processor is 110 MHz.

VectorBlox MXP can exploit the data parallelism in the computation by organizing the hardware into a tightly coupled array of the compute units that all execute the same instruction in the same cycle.

3.1.1 Scratchpad

MXP do not consist of any data or address registers. It performs all computations using scratchpad. This is used to maximize the performance as it prevents in performing the fetch and load. It does not require additional memory to hold the values.

The primary mechanism to transfer the data to the scratchpad is using the DMA (Direct Memory Access). Host processor can access the elements present in the scratchpad directly.

3.1.2 Other Processor State

MXP soft-processor consist of 32 bits control registers. First 16 registers are used for only hardware. Second group of control registers are software defined.

3.1.3 Overlapping Communication with Computation

Overlapping of the Communication with Computation is necessary to obtain maximum performance. Large set of data are processed usually by decomposing the computation into smaller blocks which further depends on the scratchpad size, and processing chunk of block simultaneously when other block is being transferred. Hence, the time required for transferring the data to the local scratchpad is hidden during the actual computation.

For example, consider the code in figure 3.2, which calculates the cube of a and puts the result in variable b. It utilizes small blocks having size of M and the dataset being used is having a size of M*N.

For improved performance and efficiency, the code can be rewritten in such a way that the computation and communication can be overlapped. Double buffering is one of the methods through which you can achieve the overlapping of computation and communication. In this technique, one buffer holds the current processing data while the other buffer job is to transfer data in and out of the local scratchpad.

Consider the code in figure 3.3 which shows the overlapping of the communication and computation. The same concept is used in measuring the performance for huge

```
/*

Code Snippet showing the overlapping of Computation and Communication

*/


vbx_word_t *v_a = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_b = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_set_vl( M );
for( i = 0; i < M*N; i += M )
{
    vbx_dma_to_vector( v_a, a+i, M*sizeof(vbx_word_t) );
    vbx( VVW, VMUL, v_b, v_a, v_a );
    vbx( VVW, VMUL, v_b, v_b, v_a );
    vbx_dma_to_host( b+i, v_b, M*sizeof(vbx_word_t) );
}
```

Figure 3.2: Overlapping of the Computation and Communication

```
/*

Code Snippet showing the overlapping of Computation and Communication using the
double buffering strategy.

*/

vbx_word_t *v_a0 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_a1 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_b0 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_b1 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_tmp;

vbx_set_vl( M );
vbx_dma_to_vector( v_a0, a, M*sizeof(vbx_word_t) );
for( i = 0; i < M*N; i += M ){
    if( i < M*N-M ){
        vbx_dma_to_vector( v_a1, a+i+M, M*sizeof(vbx_word_t) );
    }
    vbx( VVW, VMUL, v_b0, v_a0, v_a0 );
    vbx( VVW, VMUL, v_b0, v_b0, v_a0 );
    vbx_dma_to_host( b+i, v_b0, M*sizeof(vbx_word_t) );

    //Swap buffers by changing pointers
    v_tmp = v_a0; v_a0 = v_a1; v_a1 = v_tmp;
    v_tmp = v_b0; v_b0 = v_b1; v_b1 = v_tmp;
}
```

Figure 3.3: Overlapping Computation with Communication

set of input data samples in chapter 5.

3.2 Programming Model

A basic program for MXP can be written by following the procedure below:

1. Allocate the vectors in local scratchpad.
2. DMA transfer from DDR to the local scratchpad.
3. Set the vector length register. It indicates the number of vector elements on which the vector operations are to be performed.
4. Perform vector operations and obtain result.
5. Move data from the scratchpad to DDR. DMA transfer of result from local scratchpad to the DDR.

A sample example to explain the methodology is shown in Figure 3.4 The details of the programming methodology used can be found at [19]. The figure shows a sample MXP program with the overall software process followed by MXP. The program shown in Figure 3.4 works fine in a bare-metal system.

In the case when we want to run any image and audio processing application, we will need a filesystem through which information in form of bytes from the input file can be extracted without having to manually enter values as in the example above. Also, the results after processing needs to be written to a file in the appropriate file format depending on the input file type. Vectorblox do provide sources for Linux containing necessary device drivers along with prebuilt bitstreams and hardware design file generated through Vivado.

We provide detailed steps for setting up Linux OS to use MXP on the ZedBoard so that it becomes easy for others to develop application and make full use of the system. Chapter 4 consist of the detailed information on how the Linux can be set up to use MXP.

```
1  /*
2  Simple example to show the programming methodology and the overall software process
3  involved.
4  Following Examples performs the vector and constant multiplication
5  */
6
7  int input_data[512] = { 0,1,2,3, 10, 11, 12, 13, ... ,511 };
8  int multiplier = 4;
9
10 I). Allocate vectors in local scratchpad
11
12 vbx_word_t* vdata;
13 v_data = vbx_sp_malloc( 512*4 ); // 512 words,in scratchpad
14
15 II). Moving data from main memory DDR3 to local scratchpad
16
17 vbx_dcache_flush( data, 512*4 ); //remove data from cache
18 vbx_dma_to_vector( vdata, data, 512*4 );// copy from 'data'
19
20
21 III). Set the vector length of register
22
23 vbx_set_vl( 512 );// No of elements
24
25 IV). Performing vector operation
26
27 vbx( SVW, VMUL, vdata, multiplier, vdata ); //instruction
28
29 V). Moving data from local scratchpad to DDR
30
31 vbx_dma_to_host( data, vdata, 512*4 ); // copy data back
32 vbx_sync(); // wait vector/DMA to finish
33
```

Figure 3.4: Overall Software Process followed by MXP

Chapter 4

OS Support for MXP

4.1 Configuring the Linux For MXP on Xilinx ZedBoard

Vivado Design Suite 2016 along with the SDK option is installed so that cross-compiler which is required for building the Linux kernel sources is already installed. We don't require to install cross-compiler toolchain for ARM now. We plan to use a persistent filesystem rather than ramdisk since ramdisk lose changes made to filesystem when the board is powered off. Xillybus on ZedBoard along with SD card with a persistent root filesystem was made ready. Next step was the need to build FSBL (First Stage Boot Loader), device tree, U-boot, and the Linux kernel for booting up the board along with the required MXP setup. For this we need to clone the following repositories from GitHub.

1. MXP repo: `https://github.com/VectorBlox/mxp.git`
2. Linux repo: `https://github.com/VectorBlox/linux-xlnx.git`
3. U-boot repo: `https://github.com/Xilinx/u-boot-xlnx.git`
4. Device-tree repo: `https://github.com/Xilinx/device-tree-xlnx.git`

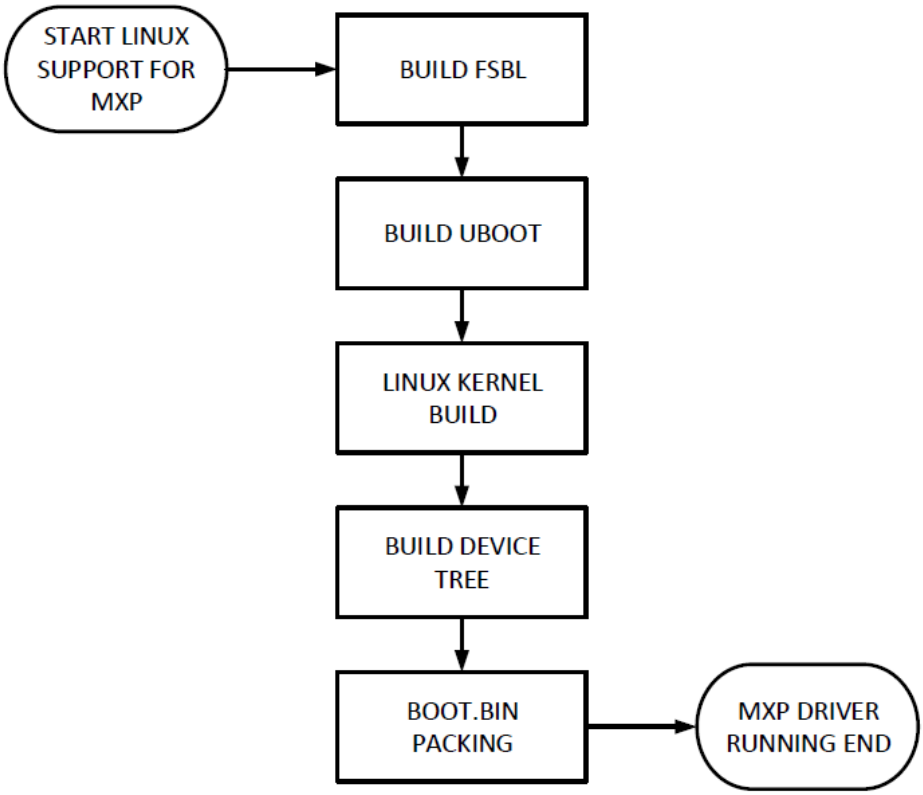


Figure 4.1: Flowchart for Linux setup

Inside the MXP repository, prebuilt bitstream (named system wrapper.bit to be instantiated on the FPGA) and hardware design file (named system.hdf) have been provided for 8 and 16 vector lanes configuration. We used the one with 16 vector lanes. Steps for setting up Linux on the ZedBoard for MXP are represented using the flowchart shown in figure 4.1 and have been explained in detail later in this chapter.

4.2 Building First Stage Bootloader

Inside the MXP repository, navigate to folder prebuilt_zedboard_arm_viv_v16 and type commands shown next. Once done rename executable.elf inside mxp_fsbl to fsbl.elf.

```
1 //hsi command helps to switch to a TCL shell
2 hsi
3 hsi% set hw_design [open_hw_design system.hdf]
4 hsi% generate_app -hw $hw_design -os standalone -proc
5 ps7_cortexa9_0 -app zynq_fsbl -compile -sw fsbl -dir mxp_fsbl
6 hsi% quit
```

4.3 Building U-BOOT

1. Navigate to u-boot repository. We plan to use persistent filesystem rather than ramdisk. Below modifications are necessary in file include/configs/zynq common.h. Find sdboot entry and edit it. This is used to avoid loading ramdisk and make the filesystem persistent.

```
1  // change the following content
2  "sdboot=echo Copying Linux from SD to RAM...;" \
3  "mmcinfo;" \
4  "fatload mmc 0 0x3000000 ${kernel_image};" \
5  "fatload mmc 0 0x2A00000 ${devicetree_image};" \
6  "fatload mmc 0 0x2000000 ${ramdisk_image};" \
7  "bootm 0x3000000 0x2000000 0x2A00000\0" \
8
9  // to the following content
10 "sdboot=echo Copying Linux from SD to RAM...;" \
11 "mmcinfo;" \
12 "fatload mmc 0 0x3000000 ${kernel_image};" \
13 "fatload mmc 0 0x2A00000 ${devicetree_image};" \
14 "bootm 0x3000000 - 0x2A00000\0" \
```

2. Then to compile u-boot give commands as shown below:

```
1 // commands to compile u-boot
2 export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
3 export ARCH=arm
4 make zynq_zed_config
5 make
```

We renamed u-boot to u-boot.elf and while building Linux kernel we need mkimage utility, so add the tools/ folder to the $PATH variable.

4.4 Linux Kernel Build

1. Navigate to Linux repository and type the following commands.

```
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
make ARCH=arm xilinx_zynq_defconfig
```

   This will generate a .config file.

2. For building the kernel supporting the MXP we need to find out the necessary configuration parameters which needs to be tweaked. To do this, we do reverse engineering. If we look at example source codes provided in the bmark (benchmark) directory of MXP repo, we see the first function to be called in these sample examples is vbx_test_init(). For these source codes to run on Linux, this function calls VectorBlox MXP Initialize ("mxp0", "cma"). If we look at the definition of this function, it uses device files devmxp0 and devcma to do some memory mapping and other initializations. Since these device files are used, there must be corresponding device drivers for them in the Linux source. We surely find files named mxp.c and cma.c in the drivers/char/ directory which are responsible for creating these device files. We then look at the Makefile in drivers/char/ directory and find entries for these files as shown below:

```
obj-$(CONFIG_MXP) += mxp.o
obj-$(CONFIG_CM_ALLOCATOR) += cma.o
```

   Now we have the configuration parameters and hence we edit the .config file generated in step 1 and set these parameters as :

```
CONFIG_MXP=m
CONFIG_CM_ALLOCATOR=y
```

3. Compile the kernel and kernel modules:

```
make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
make ARCH=arm modules
```

   This will generate compiled kernel uImage in arch/arm/boot/ directory.

4. Insert the SD card with Xillybus setup into the computer where this kernel is compiled and give:

```
make ARCH=arm modules_install INSTALL_MOD_PATH=/path/to/rootfs(
ext4partition)/in/sdcard/
```

This will copy all the kernel modules built in step 4 into the path: IN- STALL MOD PATH/lib/modules/{kernel image name}/

4.5 Building Device Tree

1. Inside Linux repo, compile device tree source (dts) to generate device tree blob (dtb).

```
// Generating the dtb
./scripts/dtc/dtc -I dts -O dtb arch/arm/boot/dts/zynq-zed.dts
 -o mxp.dtb
./scripts/dtc/dtc -I dtb -O dts mxp.dtb -o mxp_linux.dts
```

To avoid using ramdisk, we replace the contents of bootargs under the chosen node in mxp_linux.dts as shown:

```
//change this content
bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk";
//to this content
bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2
rw earlyprintk
rootfstype=ext4 rootwait devtmpfs.mount=0";
```

2. Copy folders from drivers/ directory inside MXP repo to the Device tree repo. Navigate to folder prebuilt zedboard_arm_viv_v16 in MXP repo, and enter following commands:

```

// hsi command opens up the TCL shell
hsi
hsi% open_hw_design system.hdf
hsi% set_repo_path /path/to/Device tree repo/
hsi% create_sw_design mxp_device_tree -os device_tree -proc
ps7_cortexa9_0
hsi% generate_target -dir mxp_dts
```

```
}

// DTS entry for MXP
amba_pl: amba_pl {
#address-cells = <1>;
#size-cells = <1>;
compatible = "simple-bus";
ranges ;
vectorblox_mxp_arm_0: vectorblox_mxp@b0000000 {
compatible = "xlnx,vectorblox-mxp-1.0";
reg = <0Xb0000000 0X10000 0X40000000 0X100000>;
vblx, = <1>;
vblx,archical = <0>;
vblx,beats_per_burst = <16>;
vblx,burstlength_bytes = <128>;
vblx,c_instr_port_type = <2>;
vblx,c_m_axi_addr_width = <32>;
vblx,c_m_axi_data_width = <64>;
vblx,c_m_axi_len_width = <4>;
vblx,c_m_axi_support_threads = <0>;
vblx,c_s_axi_addr_width = <32>;
vblx,c_s_axi_baseaddr = <0XB0000000>;
vblx,c_s_axi_data_width = <32>;
vblx,c_s_axi_highaddr = <0XB000FFFF>;
vblx,c_s_axi_instr_addr_width = <32>;
vblx,c_s_axi_instr_baseaddr = <0X40000000>;
vblx,c_s_axi_instr_data_width = <32>;
vblx,c_s_axi_instr_id_width = <6>;
.................................
.................................
.................................
vblx,vector_lanes = <16>;
```

After this, we see some dtsdtsi files created inside mxp_dts folder. We already have the device tree entries corresponding to various peripherals in mxp_linux.dts file created in step 1. Hence, we are only interested in pl.dtsi files which will contain the device tree node for the MXP soft processor instantiated on programmable logic. Contents in pl.dtsi is as shown in the above DTS entry for MXP code snippet.

Given the way in which the device driver file mxp.c parses this device tree node, we need to change contents of this file for the correct setup Specifically we need to change compatible string to

Also, the register entry should contain instruction address first. So, node name and reg entry should be changed to

```
1
2 vectorblox_mxp@40000000{
```

Copy the edited amba_pl node to the end of mxp_linux.dts.

3. Finally the device tree is compiled.

4.6 Packing into BOOT.bin

Create a file with name, say bootimage.bif. Paste following contents into it

```
1
2  /* Preparing the BOOT.bin package.
3     Make sure the contents are ordered as shown
4  */
5 the_ROM_image:
6 {
7  [bootloader] <path to fsbl.elf>
8  <path to bitstream file>
9  <path to u-boot.elf>
```

Save the file and generate boot.bin using the command below:

Copy files boot.bin, devicetree.dtb and uImage into the FAT partition of sdcard. Plug-in the sdcard and boot ZedBoard. While the kernel is getting loaded we should some message like this

```
1
2  ............
3  ............
4  mxp_init
5  mxp_probe
6  ............
```

This means mxp driver is successfully loaded.

4.7 Configuration Parameters of MXP

We successfully loaded the MXP drivers. The configuration parameters for the MXP setup includes some important parameters named as below:

1. vector_lanes.
2. core_freq.
3. scratchpad_size.
4. dma data width in bytes.

```
1
2  // configuration parameters for the MXP setup
3 vector_lanes = 16
4 core_freq = 100.0e6
5 scratchpad_size = 65536
```

4.8 Code for Configuration in Detail

The configuration details can be found in the below link:

Github Link : `https://github.com/AdhikariSaurabh/mxpbenchmarks/blob/master/mxplinuxsetup.doc`

Chapter 5

Performance Analysis of compute kernels

5.1 Benchmarks Evaluation

The benchmark codes present in the MXP repository provided by VectorBlox gives further idea about the kind of speedup and throughput that can be obtained using MXP soft-vector processor. Scalar time is measured for code running on ARM v7 along with NEON SIMD unit and vector time represents time required when MXP is used for acceleration. We can run the benchmark code and obtain speedup which is the ratio of the scalar time and the vector time. The benchmarks are run on Linux. We need to compile vbxapi library present in MXP repo to use MXP APIs, on Linux. Once built, this library needs to be linked while compiling the application which we will write on Linux. The results obtained are shown in Table 5.1.

Benchmark	**Scalar time (sec)**	**Vector time (sec)**	**Speedup = (Scalar time / Vector time)**
vbw_mtx_median_t	124e-3	5.012e-6	24.75
vbw_mtx_sobel	197.4e-3	24.9e-3	7.911
vbw_vec_fir_t	847e-6	58.5e-6	14.49
vbw_mtx_fir_t	10.62e-3	1.75e-3	6.057
vbw_mtx_mm_t	276e-6	18.65e-6	14.75

Table 5.1: Speedup for the benchmarks on Linux

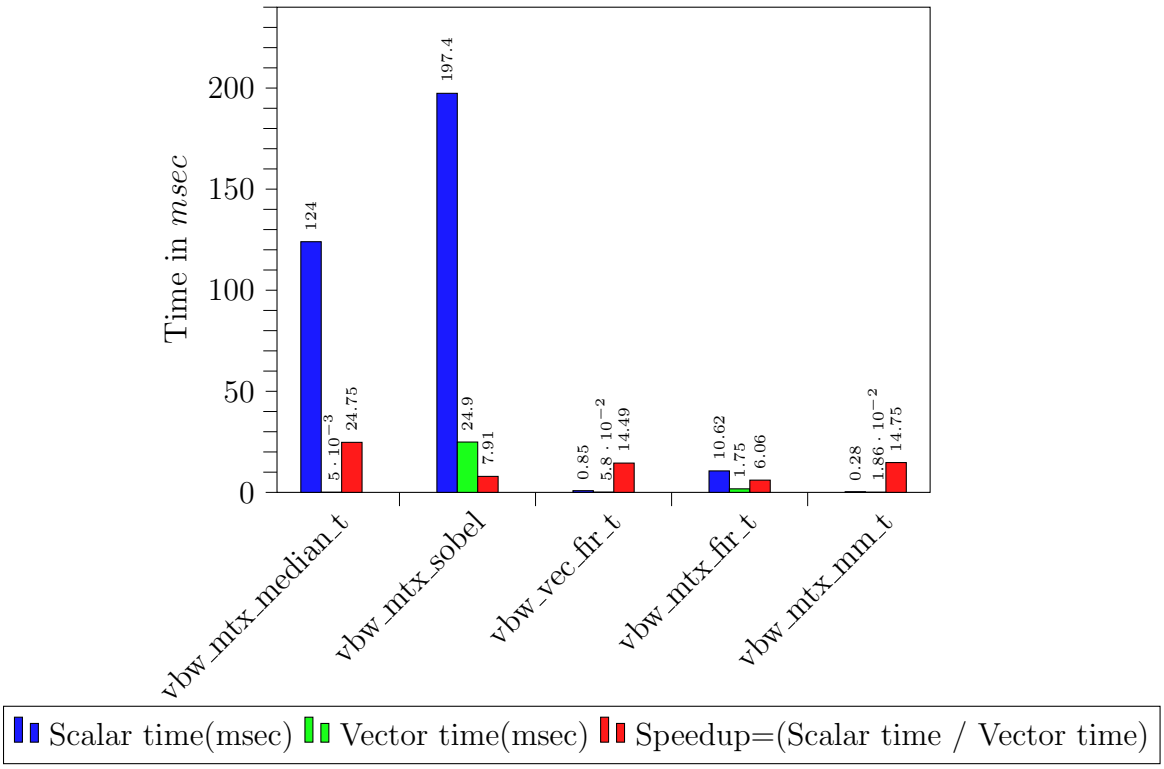


Figure 5.1: Speedup for different benchmarks

The graph showing the speedup of different benchmarks are shown in the Figure 5.1

5.2 Accelerating Compute Kernels

We have taken into consideration a benchmark set shown in the table 5.2 consisting of various compute kernels. These compute kernels are considered from the paperwork [20],[21],[22],[23].These are quite good benchmarks that can be used to analyze the performance of the MXP soft processor against other embedded processors such as ARM v7, NEON SIMD unit and INTEL i3. We took the throughput as one of the main performance measures and compared the results for various compute kernels.

The Table 5.2 demonstrates the characteristics of the benchmarks. Total number of the operations which are used to obtain the corresponding DAG (Directed Acyclic

No	Benchmark Name	Description	Add	Sub	Mul	OR	Operations
1	poly	degree-2 polynomial	2		3		5
2	poly	Degree-3 polynomial	3		6		9
3	chebyshev	filter	1	1	5		7
4	mibench	filter	8		7		15
5	sgfilter	filter	5	4	9		18
6	qspline	filter	4		23		27
7	fft	kernel	3	3	4		10
8	kmeans	kernel	7	8	8		23
9	mm	kernel	7		8		15
10	spmv	kernel	6		8		14
11	stencil	kernel	10	2	2		14
12	mri	kernel	4		5	1	10

Table 5.2: Benchmarks Characteristics

Graph) expression for compute kernels and corresponding Add, Sub, Mul and OR operations required are shown in table 5.2.

Table 5.3 demonstrates the kernel benchmarks along with the DAG expression corresponding to it. We mapped the kernel benchmarks on the MXP soft processor and obtained the corresponding throughput as it's one of the performance criterion that differentiates MXP from the other embedded hard processors.

5.3 Experimentation and Results

In our work, Xilinx ZedBoard is being used for experimentation and result analysis. We used the native gcc along with –cpu=Cortex-A9 as the compiler option for generating the ARM binaries. Auto vectorization method was used to run the code on the NEON SIMD at the highest optimization level with the compiler option to be -mfpu=neon -ftree-vectorize. We used -mfloat-abi=hard -mfpu=neon for running the code on the MXP soft-vector processor and the compute kernel performance were also analyzed running them on the Intel i3 dual core processor.

No.	Kernel	DAG Expression
1	chebyshev	$out = (x * (x *(16 * x^2 – 20) * x + 5))$
2	sgfilter	$out = (x * (x *(7 * x – 76 * y + 7) + y * (92 * y – 39) + 7) – y * (y *(984 * y + 46) + 46) – 75)$
3	mibench	$out = (x * (x + 2 * y + 6 * z + 43) + y * (y + 6 * z + 43) + z * (9 * z + 1))$
4	qspline	$out = (z * u^4 + 4 * a * u^3 * v + 6 * b * u^2 * v^2 + 4 * w * v^3 * u + q * v^4)$
5	poly-2	$out = a * x^2 + b * x + c$
6	poly-3	$out = a * x^3 + b * x^2 + c * x + d$
7	fft	$out_0 = (in_0 – (in_1 * in_2 + in_3 * in_4))$ $out_1 = (in_0 + (in_1 * in_2 + in_3 * in_4))$ $out_2 = (in_5 – (in_1 * in_4 + in_3 * in_2))$ $out_3 = (in_5 – (in_1 * in_4 + in_3 * in_2))$
8	kmeans	$out = (in_0 – in_1)^2 + (in_2 – in_3)^2 + (in_4 – in_5)^2 + (in_6 – in_7)^2 + (in_8 – in_9)^2 + (in_{10} – in_{11})^2 + (in_{12} – in_{13})^2 + (in_{14} – in_{15})^2$
9	mm	$out = (in_0 * in_1) + (in_2 * in_3) + (in_4 * in_5) + (in_6 * in_7) + (in_8 * in_9) + (in_{10} * in_{11}) + (in_{12} * in_{13}) + (in_{14} * in_{15})$
10	mri	$out_0 = (in_6 * (in_0 * in_1 + in_2 * in_3 + in_4 * in_5) + in_7) * (in_9 \mid in_{10})$ $out_1 = (in_6 * (in_0 * in_1 + in_2 * in_3 + in_4 * in_5) + in_8) * (in_9 \mid in_{10})$
11	spmv	$out_0 = (in_0 * in_1) + (in_2 * in_3) + (in_4 * in_5) + (in_6 * in_7)$ $out_1 = (in_8 * in_9) + (in_{10} * in_{11}) + (in_{12} * in_{13}) + (in_{14} * in_{15})$
12	stencil	$out_0 = (in_0 + in_1 + in_2 + in_3 + in_4 + in_5) * in_6 - in_7$ $out_1 = (in_8 + in_9 + in_{10} + in_{11} + in_{12} + in_{13}) * in_6 – in_{14}$

Table 5.3: Kernel Benchmarks

5.3.1 Runtime Comparison

We compared the runtime for the poly benchmarks (degree-2 and degree-3 polynomial) for huge number of input samples. We observed that MXP outperforms NEON SIMD unit and ARM v7 as it parallelizes the computation and communication. MXP has very high speedups due to the superior double-buffered memory transfer optimization [3.1.3]. The Table 5.4 and Table 5.5 represents the runtime in milliseconds for different number of input samples for the poly-2 (quadratic) and poly-3 (cubic) benchmarks respectively. We also plotted a graph between Time in milliseconds Vs Datasize which shows that the time taken by the MXP soft-vector processor for processing the input samples is very less and provides high speedups. Figure 5.2 and

Figure 5.3 shows the observed runtime for the poly (degree-2 and degree-3) benchmarks. This clearly infers that the MXP soft-vector processor takes less runtime as compared to ARM v7 and Neon while we keep on increasing the number of the input samples. MXP has very high speedups due to the superior double-buffered memory transfer optimization [3.1.3].

No of Samples	MXP (ms)	ARMv7 CPU (ms)	INTEL i3 (ms)	NEON(auto vector) (ms)
2^{17}	0.4239	10	0.127	-
2^{18}	0.884	10	0.262	-
2^{19}	1.88	10	0.654	10
2^{20}	3.465	20	1.057	10
2^{21}	6.949	40	4.951	10
2^{22}	13.86	70	7.962	20
2^{23}	27.72	130	9.447	40
2^{24}	55.46	250	17.275	80

Table 5.4: Poly-2 Benchmark Runtime in ms for 8-bit data

No of Samples	MXP (ms)	ARMv7 CPU (ms)	INTEL i3 (ms)	NEON(auto vector) (ms)
2^{17}	0.534	10	0.127	-
2^{18}	1.032	10	0.262	-
2^{19}	2.064	10	0.654	10
2^{20}	4.166	30	1.057	10
2^{21}	8.294	40	4.951	20
2^{22}	16.50	80	7.962	40
2^{23}	33.01	160	9.447	50
2^{24}	65.97	330	17.275	80

Table 5.5: Poly-3 Benchmark Runtime in ms for 8-bit data

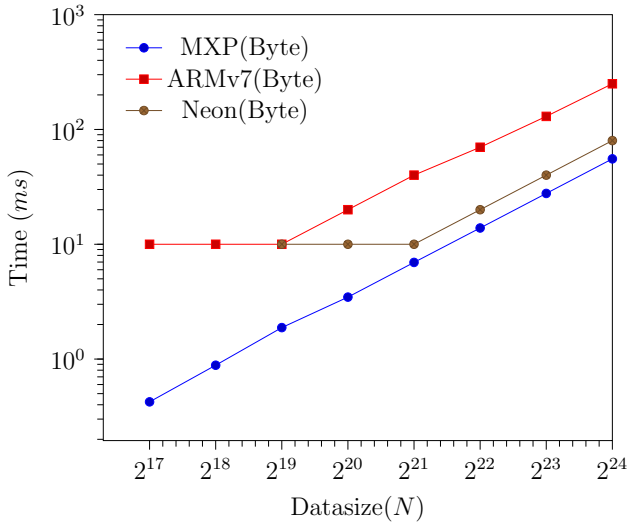
5.3.2 Benchmarks Performance Analysis

In this section, we are going to compare the MXP soft-vector processor with other embedded hard processor such as ARM v7, NEON SIMD unit and INTEL i3. The main criterion of the comparison would be the throughput expressed in terms of Gops/sec.

5.3.2.1 Throughput Analysis for poly benchmark

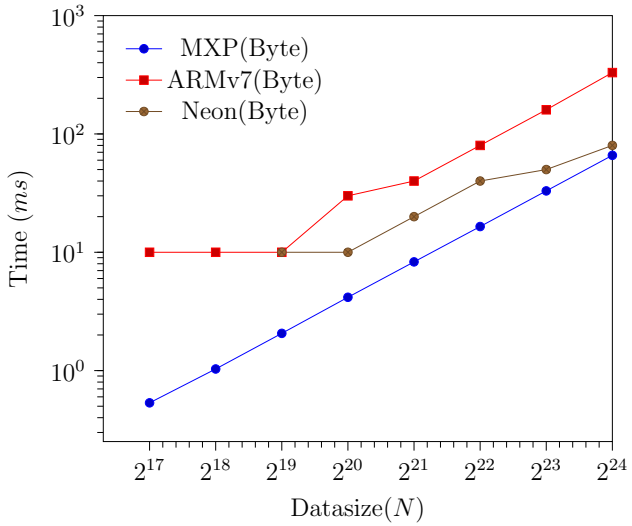
The compute kernels for the first part of performance comparison work which we took were polynomial with degree-2 (quadratic) and degree-3 (cubic). We observed

that for these benchmarks, MXP can provide a maximum throughput of 1.51 Gops for quad and 2.287 Gops for the cubic kernel when it is operating at the byte(8-bits) level. These results shows that the MXP soft-processor is better than ARMv7 (4.75×)



$$f(samples, time) = a * x^2 + b * x + c$$

Figure 5.2: Comparing Runtime for *Poly-2* benchmark.



$$f(samples, time) = a * x^3 + b * x^2 + c * x + d$$

Figure 5.3: Comparing Runtime for *Poly-3* benchmark.

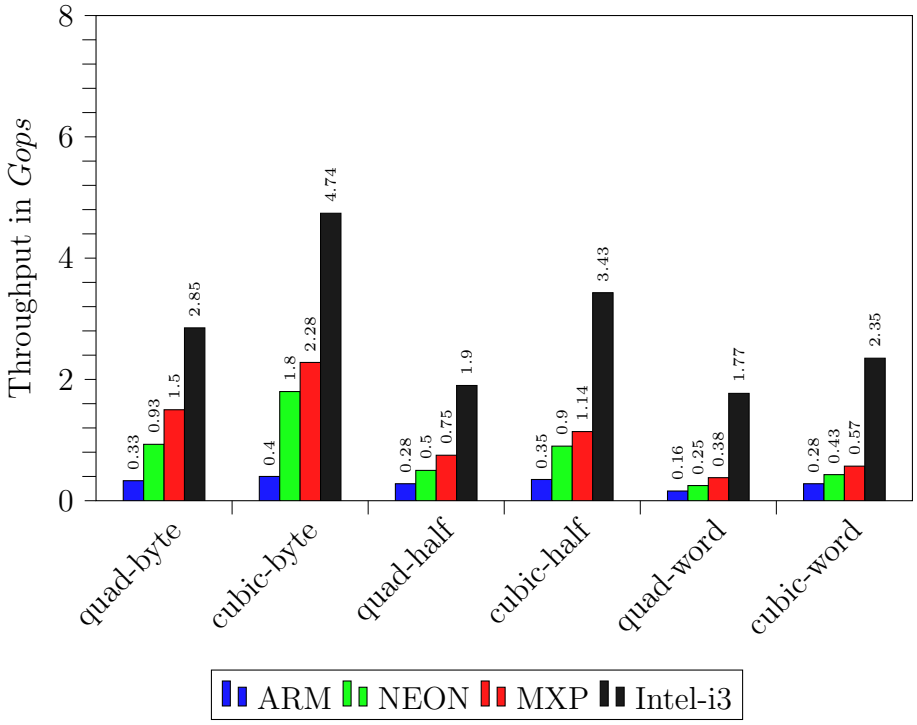


Figure 5.4: The performance comparisons of different architectures for quad and cubic

and NEON (1.56×) while operating on quadratic kernel and is better than ARMv7 (5.71×) and NEON (1.34×) while operating on cubic kernel. MXP provides better speedups due to the superior double-buffered memory transfer optimization [3.1.3]. Architecture specification and the corresponding performance in terms of throughput for the poly (degree-2 and degree-3) benchmarks are mentioned in table 5.6. Figure 5.4 demonstrates the average throughput (Gops/sec) obtained while operating on the byte, halfword and word level. It can be inferred from the values that the MXP soft processor provides highest speedups when it operates at the byte level. It provides a speedup of (4×) when it operates at byte level and (2×) when it operates at halfword level when compared to the word level speedup.

5.3.2.2 Throughput Analysis for filter compute kernel

The compute kernels for second part of performance comparison which we took were the standard filter kernel. Throughput analysis for filter compute Kernels such as

Metrics		**ARMv7 CPU**	**NEON (auto vector)**	**MXP**	**INTEL i3**
arch		Scalar	SIMD unit	Vector	Scalar
clock		$667X10^6Hz$	$667X10^6Hz$	$110X10^6Hz$	$2X10^9Hz$
no of lanes		1	2 x 32b 4 x 16b 8 x 8b	1-16 x 32b 2-32 x 16b 4-64 x 8b	1
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	Quadratic- Poly Deg 2	0.1608	0.2500	0.3782	1.77
16b(Gops/sec)/Halfword		0.2819	0.4988	0.758	1.914
8b(Gops/sec)/Byte		0.33656	0.9341	1.513	2.85
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	Cubic- Poly Deg 3	0.2886	0.4311	0.571	2.35
16b(Gops/sec)/Halfword		0.356	0.893	1.144	3.43
8b(Gops/sec)/Byte		0.40054	1.7816	2.287	4.74

Table 5.6: Throughput(Gops/sec) Analysis for Poly Benchmarks

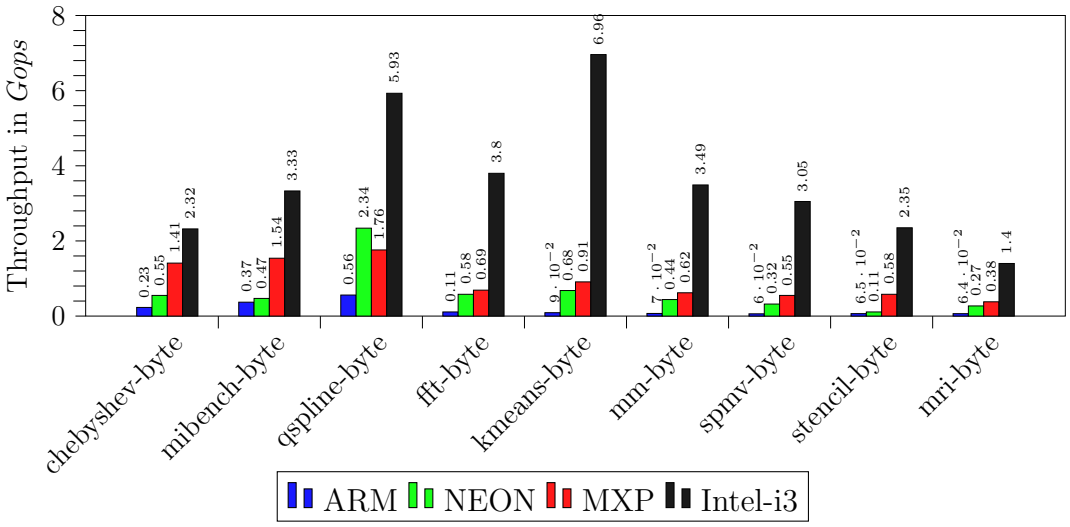


Figure 5.5: Byte Level performance comparisons of different architectures for kernels

Chebyshev, Mibench, Sgfilter and Qspline were done. We observed that for these benchmarks, MXP have better throughput when it is operating at the byte(8-bits) level. MXP can provide a maximum throughput of 1.41 Gops for chebyshev kernel, 1.54 Gops for the mibench kernel and 1.76 Gops for the qspline kernel when it is operating at the byte(8-bits) level. These results shows that the MXP soft-processor is better than ARMv7 (6.38×) and NEON (2.56×) while operating on chebyshev kernel and while operating on mibench kernel is better than ARMv7 (4.21×) and

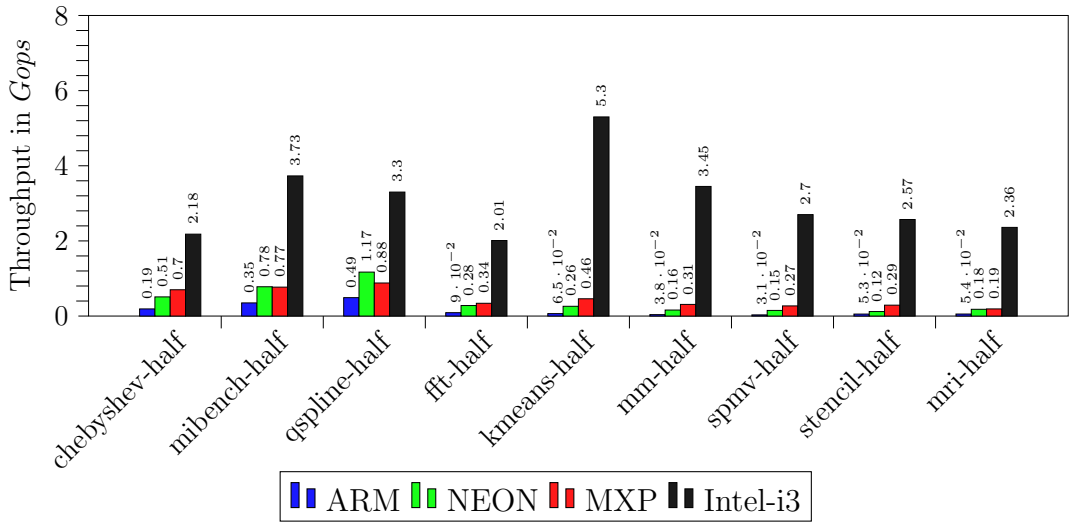


Figure 5.6: Halfword performance comparisons of different architectures for kernels

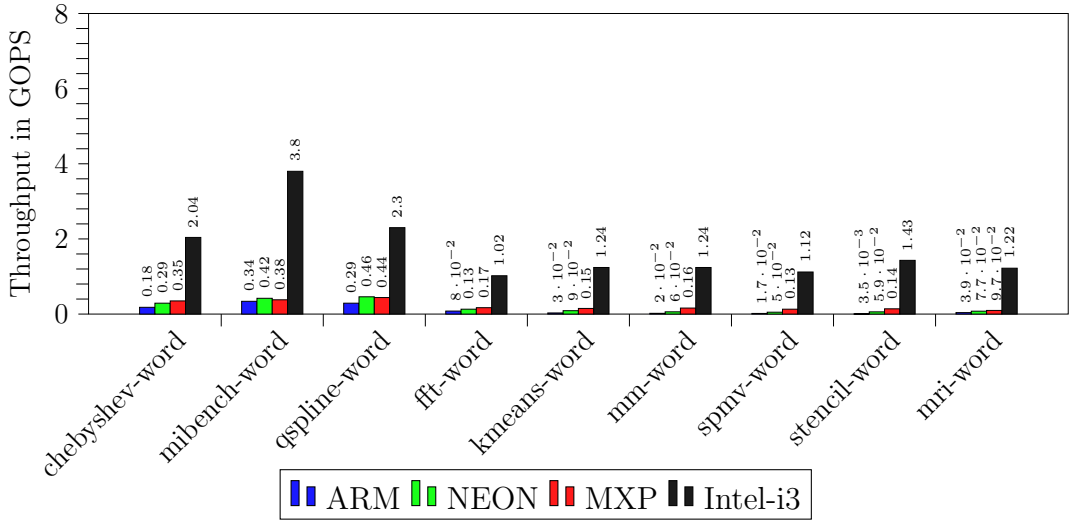


Figure 5.7: Word Level performance comparisons of different architectures for kernels

NEON (3.34×) . MXP has very high speedups due to the superior double-buffered memory transfer optimization [3.1.3].

Table 5.7 explains the architecture specification and the corresponding throughput obtained in terms of Gops/sec for various benchmarks. Figure 5.5, 5.6 and 5.7 demonstrates the average throughput (Gops/sec) obtained while operating on the word (32-bits), halfword (16- bits) and byte (8-bits) levels. We also observed that

for a specific benchmark, the performance at the byte level was 4 times and at the halfword level was 2 times when compared with the performance at the word level for the MXP soft-vector processor. For MXP, we have the width of the data as selectable parameter.

Metrics		**ARMv7 CPU**	**NEON (auto vector)**	**MXP**	**INTEL i3**
arch		Scalar	SIMD unit	Vector	Scalar
clock		$667X10^6Hz$	$667X10^6Hz$	$110X10^6Hz$	$2X10^9Hz$
no of lanes		1	2 x 32b 4 x 16b 8 x 8b	1-16 x 32b 2-32 x 16b 4-64 x 8b	1
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	CHEBYSHEV	0.1832	0.296	0.354	2.040
16b(Gops/sec)/Halfword		0.1955	0.5159	0.7085	2.186
8b(Gops/sec)/Byte		0.2219	0.5585	1.41	2.324
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	MIBENCH	0.3478	0.426	0.386	3.816
16b(Gops/sec)/Halfword		0.352	0.7831	0.773	3.738
8b(Gops/sec)/Byte		0.3659	0.460	1.547	3.338
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	QSPLINE	0.298	0.4641	0.442	2.372
16b(Gops/sec)/Halfword		0.493	1.169	0.88	3.38
8b(Gops/sec)/Byte		0.561	2.341	1.76	5.935

Table 5.7: Throughput(Gops/sec) Analysis for Filter Benchmarks

5.3.2.3 Throughput Analysis for Standard kernels

As a third part of performance comparison work, we took various benchmarks such as FFT, KMEANS, MM, SPMV, STENCIL and MRI which are the standard compute kernels. We observed that for these compute kernels, MXP has high throughput. The maximum throughput that can be obtained for the MXP soft-processor at byte level for fft kernel is 0.69 Gops, kmeans is 0.91 Gops, mm is 0.62 Gops, spmv is 0.55 Gops, stencil is 0.59 and for mri it is 0.38 Gops. MXP has better speedups due to the superior double-buffered memory transfer optimization [3.1.3]. Table 5.8 explains the architecture specification and the corresponding throughput obtained in terms of Gops/sec for various benchmarks. Figure 5.5, 5.6and 5.7 demonstrates the average throughput (Gops/sec) obtained while operating on the word (32-bits), halfword (16-

bits) and byte (8-bits) levels.

Metrics		**ARMv7 CPU**	**NEON (auto vector)**	**MXP**	**INTEL i3**
arch		Scalar	SIMD unit	Vector	Scalar
clock		$667X10^6Hz$	$667X10^6Hz$	$110X10^6Hz$	$2X10^9Hz$
no of lanes		1	2 x 32b 4 x 16b 8 x 8b	1-16 x 32b 2-32 x 16b 4-64 x 8b	1
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	FFT	0.0867	0.137	0.174	1.0283
16b(Gops/sec)/Halfword		0.0937	0.2887	0.349	2.01
8b(Gops/sec)/Byte		0.1042	0.5833	0.6995	3.809
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	KMEANS	0.03174	0.09083	0.2285	1.918
16b(Gops/sec)/Halfword		0.06582	0.257	0.457	5.301
8b(Gops/sec)/Byte		0.0986	0.689	0.9143	6.96
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	MM	0.02015	0.06009	0.156	1.24
16b(Gops/sec)/Halfword		0.03806	0.165	0.3136	3.456
8b(Gops/sec)/Byte		0.0667	0.443	0.627	3.495
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	SPMV	0.01680	0.0532	0.1399	1.126
16b(Gops/sec)/Halfword		0.0307	0.1529	0.279	2.696
8b(Gops/sec)/Byte		0.061	0.391	0.5596	3.05
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	STENCIL	0.03589	0.05866	0.14	1.437
16b(Gops/sec)/Halfword		0.0534	0.1146	0.29	2.57
8b(Gops/sec)/Byte		0.0651	0.1162	0.589	2.35
Throughput	**Benchmark**				
32b(Gops/sec)/ Word	MRI	0.039	0.077	0.097	1.221
16b(Gops/sec)/Halfword		0.0537	0.183	0.194	2.36
8b(Gops/sec)/Byte		0.064	0.277	0.388	1.406

Table 5.8: Throughput(Gops/sec) Analysis for Standard Compute Kernels

5.3.2.4 Throughput Analysis for Linear Algebra Kernel

We accelerate two linear algebra kernels namely atax and bicg using MXP overlay. We change the default data type which is double to integer for the computations to have proper comparison analysis about the speedup. We accelerate the kernel for a small dataset with size of matrix 500x500 and for standard dataset size of matrix 4000x4000. The speedup obtained using MXP is as shown in the below table 5.9. We observed that MXP provide high speedup for small as well as standard dataset. For the BiCG kernel, MXP provide speedup of 3 for the small dataset and 3.25 for the

standard dataset when compared with the embedded hard processor ARM v7. For ATAX compute kernel, speedup was observed to be 5.62 for the small dataset and 4.63 for standard dataset.

Metrics		**ARMv7 CPU**	**MXP**	**NEON(auto vector)**
uarch		Scalar	Soft vector	SIMD unit
Clock		667 Mhz	110 Mhz	667 Mhz
No of Lanes		1	1-16 X32b	2 X32b
			2-32 X16b	4 X16b
			4-64 X8b	8 X8b
Runtime (ms)	**Benchmark**			
500 X 500 (small dataset)	BiCG	5.935	1.972	5.89
4000 X 4000 (standard dataset)		234.69	72.033	216.26
Runtime (ms)	**Benchmark**			
500 X 500 (small dataset)	ATAX	6.598	1.172	6.581
4000 X 4000 (standard dataset)		486.96	105.303	483.99

Table 5.9: Throughput(Gops/sec) Analysis for Linear Algebra kernel

5.4 Compute Kernel Code

Runtime and Throughput calculation application for all the compute kernels can be found from the below given repository:

Github Link : `https://github.com/AdhikariSaurabh/mxpbenchmarks/tree/master/polyresults`

Chapter 6

Accelerating Image Processing and SpMV

6.1 Basic Image Processing Application

We need to compile vbxapi library present in MXP repository to use MXP APIs, on Linux. Once built, this library needs to be linked while compiling the application which we will write on Linux. To start with, we built application for image negation and scaling operations. The input and processed image obtained after running the image negation application are shown in Figure 6.1

The MXP application code for processing the input image for a negation operation is provided in Figure 6.2

6.2 Experiments and Results

We took images of different dimensions and tried building image negative application for different platforms such as ARM v7, MXP and SIMD NEON unit. We tested the application with 128 X 128 pixels and 256 X 256 pixels. The runtime for the MXP soft-vector processor was then compared with the other processors. We observed that MXP seems to outperform other processors with a very good runtime obtained. Table 6.1 shows the runtime analysis of the MXP against the other processors.

Figure 6.1: Results of the Image Negation

Metrics	**ARMv7 CPU**	**NEON (auto vector)**	**MXP**
arch	Scalar	SIMD unit	Vector
clock	$667X10^6Hz$	$667X10^6Hz$	$110X10^6Hz$
no of lanes	1	2 x 32b	1-16 x 32b
		4 x 16b	2-32 x 16b
		8 x 8b	4-64 x 8b
Runtime (milliseconds)			
128 x 128 pixels	0.406	0.424	0.03686
256 x 256 pixels	1.714	1.198	0.1843

Table 6.1: Image Processing Runtime Analysis

6.3 Accelerating the SpMV Computational Kernel

Sparse matrix dense vector also known as SpMV is one of the widely used computational kernel. It finds its application to solve the sparse linear system in information retrieval and many other applications. Since it is widely used, it is important to have a benchmark for SpMV. However, SpMV is really very difficult to benchmark because the data structure which is used in sparse matrix, it's density of the non-zero entries and it's dimensions - all have a significant impact on SpMV performance [24],[25]. We present a method on how the pre-existing benchmark for SpMV can be used and how MXP soft-processor can be used to accelerate the SpMV.

```
1  //MXP application for Image negation
2  int main(int argc, char *argv[])
3  {
4  pgm_t opgm;
5  pgm_t ipgm;
6  int img_size = 0;
7  unsigned char *v_sub = NULL;
8  unsigned char max_val = 255;
9  // MXP initialization
10 VectorBlox_MXP_Initialize("mxp0","cma");
11 //Reading input image
12 readPGM(&ipgm,"input_lena.pgm");
13 img_size = (ipgm.width * ipgm.height);
14  //Allocate buffer for output image
15 opgm.width = ipgm.width;
16 opgm.height = ipgm.height;
17 opgm.buf = (unsigned char*)vbx_shared_malloc(img_size * sizeof(unsigned char));
18  //Allocate vector on scratchpad
19 v_sub = (unsigned char *)vbx_sp_malloc(img_size * sizeof(unsigned char ));
20 //Transfer input bytes from memory to scratchpad
21 vbx_dma_to_vector(v_sub, ipgm.buf, img_size);
22 vbx_set_vl(img_size);
23 //Scalar vector subtraction for taking negative of image
24 vbx(SVBU, VSUB, v_sub, max_val, v_sub);
25 //Writing result from scratchpad to memory
26 vbx_dma_to_host(opgm.buf, v_sub, img_size);
27 vbx_sync();
28 //Writing output image
29 writePGM(&opgm,"out_lena_negative.pgm");
30 //Free allocated pointers
31 vbx_sp_free();
32 vbx_shared_free(ipgm.buf);
33 vbx_shared_free(opgm.buf);
34 }
35
```

Figure 6.2: MXP application for Image negation

6.3.1 SpMV basics

In the case of a dense matrix, simple contiguous array is used as the data structure containing all the entries. Two orderings are most common:

1. row- major, where the rows are stored contiguously.

2. column-major, where the columns are stored contiguously.

The sparse multiplication depends on the structure of the sparse matrix. We can have regular and irregular structure for the SpMV. The general idea of performing a

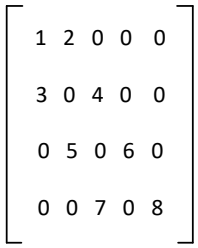


Figure 6.3: Compressed Sparse Row Format

SpMV is to decompose the matrix into regular and the irregular parts. Compressed Sparse Row Format and Compressed Sparse Column Format are the basic formats in which the sparse matrix are stored.

6.3.1.1 Compressed Sparse Row Format

In CSR format, all the nonzero entries of the matrix will be in the form of row-major order consisting of two different arrays namely row_start and col_idx. The row_start consist of index of non-zero entry present in the matrix in each row and the col_index consist of the index of the non-zero entry present in the matrix in each column. Figure 6.3 shows how the matrix elements will be stored while using the CSR format.

6.3.1.2 Compressed Sparse Column Format

In CSC format, all the nonzero entries of the matrix are kept in form of column major order consist-ing of two arrays namely col_start and row_idx. The col_index consists of index of element of each row present and row_idx consist of the index of the non-zero entry present in each row of matrix. The figure shows how the matrix elements will be stored while using the CSR format. When above Figure 6.4 is stored in Compressed Sparse Row (CSR), it will be represented in the following form

values = [1, 3, 2 , 5, 4, 7, 6, 8]

col_start = [0, 2, 4, 6, 7, 8]

row_idx = [0, 1, 0, 2, 1, 3, 2, 3]

Figure 6.4: Compressed Sparse Column Format

6.3.2 Details of Pre-existing SpMV Benchmark

The Pre-existing benchmarking framework mentioned in [24], stores the elements of the matrix in BCSR (Blocked Compressed Sparse Row) format. The idea behind the BCSR is to divide a given sparse matrix of size m x n into blocks of size r x c. By doing so we would obtain m / r blocks of rows and n / c blocks of columns which further can be stored in CSR format. Hence, at the end we will obtain m / r blocks where each of them represents r number of rows of the matrix and n / c blocks where each of them represents c number of columns of the matrix. This method helps in improving the accessing the elements of the sparse matrix and efficiently storing them in contiguous array either in row order or column order form. The operations are performed very efficiently providing high speedups. The blocks inside the dense matrix in Figure 6.5 are coloured and row major ordering is being used.

6.3.3 SpMV Usage Parameters

When the main application will run in the existing framework it will generate the following usage parameters. User have the freedom to select one of the pre-defined block size. Below are the results which were obtained when Block Dim (r x c) was by default set to 1 x 1. Table 6.2 shows the list of the SpMV usage parameters.

This pre-existing benchmarking framework consist of the block routine of dimensions {2 x 2}, {4 x 4}, {8 x 8}... Based on the block size we divide the sparse matrix into different blocks. This gives some hint to accelerate this benchmarking framework for SpMV by changing these block routines using the MXP APIs.

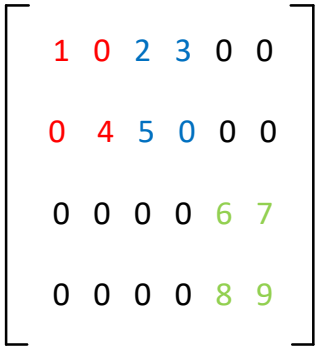


Figure 6.5: Blocked Compressed Sparse Row ordered

Parameters	**Usage**
SpMV time	0.886775
Oper Time	0.886774
Cache time	0.000001
Matrix Dim (r x c)	2097152 x 2097152
Block Dim (r x c)	1 x 1
Non-zero blocks	71303168
Repetitions	1
Mflop/s	160.814565
num_loads	293601308
num_stores	570425344

Table 6.2: SpMV Usage

6.3.4 Accelerating SpMV Framework Using MXP APIs

MXP APIs are configured for the existing benchmarking framework so that we can use these APIs to accelerate the benchmarking process of the SpMV. There are block routines which are present in the existing framework. These block routines operate on the given sparse matrix at the block level. Thus, using the MXP APIs the block routines are modified to improve its performance. Table 6.3 shows the speedup obtained while performing SpMV multiplication which is in the form of $y = Ax$, one of the widely used computational kernel where A is a Sparse Matrix and x is the input vector and output y is the dense matrix. Result analysis shows that while performing SpMV product the MXP soft processor can be faster than ARMv7 (2.07×).

Metrics	**ARMv7 CPU**	**MXP**
uarch	Scalar	Vector
Clock	$667X10^6Hz$	$110X10^6Hz$
No of Lanes	1	1-16 x 32b 2-32 X16b 4-64 X8b
Runtime (millisecond)		
SpMV {25 x 25} kernel	0.037	0.01843
SpMV {50 x 25} kernel	0.055	0.03686

Table 6.3: Runtime for SpMV kernel Computation

6.4 SpMV Kernel Code

The algorithm along with the code for calculating the runtime for SpMV kernel Computation can be found in the below link:

Github Link : `https://github.com/AdhikariSaurabh/mxpbenchmarks/tree/master/SpMV`

Chapter 7

Conclusion and Future Work

7.1 Conclusions

This thesis focuses upon using FPGA overlay architecture as an accelerator for accelerating different standard benchmarks. Configuring the Linux for MXP on the ZedBoard to provide the OS support for MXP is among one of the work done in this thesis. The performance, speedup and runtime analysis of different compute kernels are obtained using the MXP overlay as a FPGA accelerator. Moreover, MXP performance is compared with different embedded hard processors such as ARM v7, NEON SIMD unit and INTEL i3. The performance in terms of throughput was measured at the byte, halfword and word level.

We accelerated Poly-2 and Poly-3 benchmarks where we could get a very high throughput as well as speedup using the MXP overlay. The results show that MXP provides speedup of ≈4.5 times the speedup provided by ARM v7 and speedup of ≈1.5 times the speedup provided by NEON SIMD unit. Throughput obtained by the MXP while accelerating poly benchmarks was 1.5130 (Gops/sec) for poly-2 and 2.287 (Gops/sec) for poly-3 benchmark which is higher as compared to throughput obtained by other processors. The above results are mentioned for the word (32-bits) level operation whereas this thesis covers the throughput measurement at byte (8-bits) and halfword (16-bits) level also.

Filter kernels such as CHEBYSHEV, QSPLINE and MIBENCH were accelerated

using MXP. For CHEBYSHEV, we could get throughput of 1.41 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. For MIBENCH, we could get throughput of 1.54 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. For QSPLINE, we could get throughput of 1.76 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. The above results are mentioned for the word (32-bits) level operation whereas this thesis covers the throughput measurement at byte (8-bits) and halfword (16-bits) level also.

Standard kernels such as FFT, KMEANS, MM, SPMV, STENCIL and MRI were also accelerated using MXP. For FFT, we could get throughput of 0.699 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. For KMEANS, we could get throughput of 0.9143 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. For MM, we could get throughput of 0.62 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. For SPMV, we could get throughput of 0.55 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. For STENCIL, we could get throughput of 0.589 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. For MRI, we could get throughput of 0.388 (Gops/sec) which is greater than SIMD NEON unit and ARM v7 processor. The above results are mentioned for the word (32-bits) level operation whereas this thesis covers the throughput measurement at byte (8-bits) and halfword (16-bits) level also.

Polybench kernels, ATAX and BiCG were accelerated using MXP. For ATAX, we could get speedup of $\approx$5.62 for small dataset size and $\approx$4.63 for standard dataset size. For BiCG, we could get speedup of $\approx$3 for small dataset size and $\approx$3.25 for standard dataset size.

We also accelerated an image processing application using the MXP overlay and measured the runtime. The time took by the MXP was 0.03686 milliseconds for image having dimension as 128 X 128 pixels and 0.1843 milliseconds for image having dimension as 256 X 256 pixels. The speedup provided by the MXP overlay was very high when compared with the speedup provided by ARM v7, NEON SIMD unit and INTEL i3 processors.

Result analysis shows that while operating on SpMV product computational kernel

the MXP soft processor can be faster than ARMv7 (2.07×).

7.2 Future Work

7.2.1 Power Analysis

In our work, focus was more on throughput (Gops/sec) as one of the performance parameter. We can run different MXP applications and measure the power dissipated. The power rails of the ZedBoard gives some hints of measuring the power for the application running on the ZedBoard.

7.2.2 Audio Processing Application

Building an audio processing application and accelerating it using the MXP overlay. The performance analysis for the audio processing application and its comparison with the other embedded hard processors like ARM v7, SIMD NEON unit and INTEL i3.

7.2.3 Completion of the SpMV Benchmarking Framework

The current existing SpMV benchmarking framework consist of different block routines that divide the input sparse dense matrix into different blocks and then process it i.e. BCSR (Blocked Compressed Sparse Row). The MXP APIs can be used to build and change the entire block routines to accelerate the SpMV computational kernel.

Bibliography

[1] Nachiket Kapre, *Optimizing Soft Vector Processing in FPGA-based Embedded Systems*, http://nachiket.github.io/publications/soft-vector_trets2016.pdf

[2] Michael Feldman and Addison Snell, *ACCELERATED COMPUTING: A TIPPING POINT FOR HPC.* http://images.nvidia.com/content/pdf/tesla/accelerated-computing-at-a-tipping-point.pdf

[3] Davor Capalija and Tarek S. Abdelrahman,*FPGA Overlays for High Performance Computing.* https://www.eee.hku.hk/~hso/olaf2013/abdelrahman_olaf.pdf

[4] Kevin Morris, *FPGA Wars: Its Getting Hot at the Top.* http://www.eejournal.com/archives/articles/20130305-fpgawars

[5] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell,*Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform*, 2Journal of Signal Processing Systems, 77(12):6176, Oct. 2014

[6] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam,*Design, integration and implementation of the dyser hardware accelerator into opensparc*, In In- ternational Symposium on High Performance Computer Architecture (HPCA), pages 112, 2012

[7] Neil W. Bergmann, Sunil K. Shukla, and Jrgen Becker,*QUKU: a dual-layer reconfigurable architecture*, ACM Transactions on Embedded Computing Systems (TECS), 12:63:163:26, March 2013

[8] A. Severance and G. G. F. Lemieux,*Embedded supercomputing in fpgas with the vectorblox mxp matrix processor*, In Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on, pages 110, 2013

[9] Xilinx Ltd. Zynq-7000 technical reference manualhttp://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf 2013

[10] http://www.nxp.com/assets/documents/data/en/application-notes/AN4991.pdf

[11] http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[12] Xilinx Ltd. Zynq-7000 technical reference manualhttp://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[13] Gordon Brebner,*A virtual hardware operating system for the Xilinx XC6200*, In Field-Programmable Logic Smart Applications, New Paradigms and Compilers, pages 327336. 1996

[14] C. Steiger, H. Walder, and M. Platzner,*Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks*, IIEEE Transactions on Computers, 53(11):13931407, November 2004

[15] C. Aws Ismail,*Operating system abstractions of hardware accelerators on field-programmable gate arrays*, Thesis, August 2011

[16] Xillybus Ltd. Xillybus: IP Core Product Briefhttp://xillybus.com/downloads/xillybus_product_brief.pdf

[17] http://www.wiki.xilinx.com/Linux

[18] http://vectorblox.github.io/mxp/mxp_quickstart_vivado.html

[19] http://vectorblox.github.io/mxp/mxp_reference.html

[20] S. Gopalakrishnan, P. Kalla, M. B. Meredith, and F. Enescu,*Finding linear building-blocks for RTL synthesis of polynomial datapaths with fixed-size bit-vectors*, ICCAD, 2007

[21] Abhishek Kumar Jain, Douglas L. Maskell,Suhaib A. Fahmy, *Throughput Oriented FPGA Overlays Using DSP Blocks.* https://www2.warwick.ac.uk/fac/sci/eng/staff/saf/publications/date2016-jain.pdf

[22] Soh Jun Jie and N. Kapre,*Comparing soft and hard vector processing in FPGA-based embedded systems*, 2014 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, 2014, pp. 1-7. doi: 10.1109/FPL.2014.6927467

[23] Nachiket Kapre,*Optimizing Soft Vector Processing in FPGA-Based Embedded Systems*, ACM Transactions on Reconfigurable Technology and Systems, vol. 9, pp. 1, 2016, ISSN 19367406

[24] Hormozd Benjamin Gahvari, *Benchmarking Sparse Matrix-Vector Multiply.* https://pdfs.semanticscholar.org/8abd/d65f91a25d724b393f5ff07cdd3f4b033468.pdf

[25] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, Katherine A. Yelick, *Performance Modeling and Analysis of Cache Blocking in Sparse Matrix Vector Multiply.* https://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/CSD-04-1335.pdf