The Unified Floating Point Vector Co-processor for Reconfigurable Hardware

A Thesis Presented

by

Jainik Kathiara

The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Master of Science
in
ELECTRICAL AND COMPUTER ENGINEERING

Advisor:

Prof. Miriam Leeser

Dissertation Committee:

Prof. Stefano Basagni
and
Prof. Ningfang Mi

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

NORTHEASTERN UNIVERSITY

Boston, Massachusetts
2010

NORTHEASTERN UNIVERSITY
Graduate School of Engineering

Thesis Title: **The Unified Floating Point Vector Co-processor for Reconfigurable Hardware**

Author: Jainik Kathiara.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser Date

Thesis Committee Member: Prof. Stefano Basagni Date

Thesis Committee Member: Prof. Ningfang Mi Date

Department Chair: Prof. Ali Abur Date

Graduate School Notified of Acceptance:

Dean: Dr. Sara Wadia-Fascetti Date

ABSTRACT

There has been an increased interest recently in using embedded cores on FPGAs. Many of the applications that make use of these cores have floating point operations. Due to the complexity and expense of floating point hardware, these algorithms are usually converted to fixed point operations or implemented using floating-point emulation in software. As the technology advances, more and more homogeneous computational resources and fixed function embedded blocks are added to FPGAs and hence implementation of floating point hardware becomes a feasible option.

In this research we have implemented a high performance, autonomous floating point vector co-processor (FPVC) that works independently within an embedded processor system. We have presented a unified approach to vector and scalar computation, using a single register file for both scalar operands and vector elements. The Hybrid vector/SIMD computational model of FPVC results in greater overall performance for most applications along with improved peak performance compared to other approaches. By parameterizing vector length and the number of vector lanes, we can design an application specific FPVC and take optimal advantage of the FPGA fabric. For this research we have also initiated designing a software library for various computational kernels, each of which adapts FPVC's configuration and provide maximal performance. The kernels implemented are from the area of linear algebra and include matrix multiplication and QR and Cholesky decomposition. We have demonstrated the operation of FPVC on a Xilinx Virtex 5 using the embedded PowerPC.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Professor Miriam Leeser for her support and encouragement during my Master's thesis and invaluable guidance for my curriculum. I would also like to express my sincere gratitude to her for patiently waiting and correcting my thesis. I feel lucky to have such a wonderful academic and research advisor, to whom you can always look up on for guidance, inspiration and patience.

I would also like to express my gratitude to Professor Stefano Basagni and Professor Ningfang Mi for agreeing to serve in my master's thesis defense committee. I would also like to thank my fellow visiting researcher Paolo Palana, who helped me to design the linear algebra kernels for benchmarking my design and providing critical review about my design. I would also like to thank my fellow researchers for their help and support throughout my research at Reconfigurable Computing Laboratory.

Finally, I would like to thank my loving family and friends for their constant support and encouragement.

Contents

List of Tables

List of Figures

1 INTRODUCTION

Reconfigurable hardware bridges a gap between ASICs (Application Specific Integrated Circuits) and microprocessor based systems. Recently, there has been an increased interest in using reconfigurable hardware for multimedia, signal processing and other computationally intensive embedded processing applications. These applications perform floating point arithmetic computation for high data accuracy and high performance. Reconfigurable hardware allows the designer to customize the computational units in order to best match application requirements and at the same time, optimize device resource utilization. Because of these advantages, extensive research has been done to efficiently implement floating point computations on the reconfigurable hardware. Floating point (FP) computations can be categorized in three classes:

1. Software library
2. General purpose floating point unit
3. Application specific floating point data path

A Software library, is easy to implement but usually results in the highest execution time. Custom hardware accelerators are highly optimized for performance but require hardware design knowledge and long design cycles. These are the two extreme corners of the design tradeoffs. General purpose floating point unit has a onetime long design cycle cost but can provide comparable performance to hardware accelerators.

Typically, FP applications have a relatively small set of operations that are repeatedly performed over a large volume of floating point data. This form of parallelism is referred to as *data-level parallelism*. For these applications, vector processing offers simple and straightforward parallelism by executing mathematical operations on multiple data elements simultaneously. Because of the availability of a large amount of heterogeneous resources on a Field Programmable Gate Array (FPGA), Yiannacouras [5] and Yu [6] have implemented two of the earliest Field Programmable Gate Array (FPGA) based vector processors and discussed various different parameterization of the processors. But these vector processors and parameterization are limited to integer arithmetic only.

Initial sections of the chapter give the background information about reconfigurable architecture, floating point data format and various FPGA based FP libraries. The following section describes related work in the area of general purpose floating point unit implementation. At the end of this chapter, we list the contributions of this research and provide an outline of the rest of the thesis.

1.1 Reconfigurable Architecture

Reconfigurable architectures is a computer architecture combining some of the flexibility of software with the high performance of hardware by processing with very flexible high speed computing fabrics. An example is Field Programmable Gate Arrays (FPGAs). The principle difference between reconfigurable architecture and a microprocessor based system is the ability to make substantial changes to the datapath itself in addition to the control flow. Similarly, reconfigurable architecture differentiates itself from ASICs by providing run time or compile time reconfigurability to adapt application specific hardware requirements and hence reduces design cycle time as well as non-recurring engineering cost.

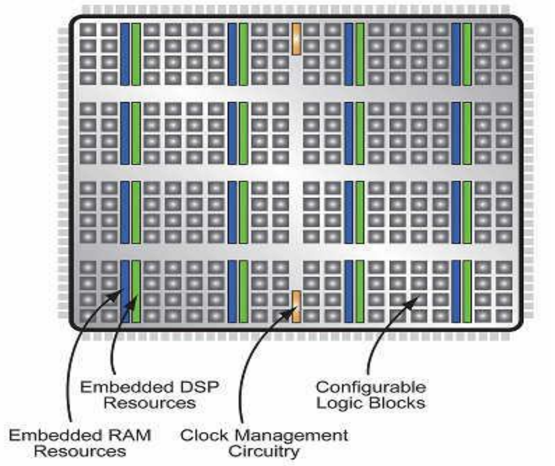


Figure 1.1 Modern Field Programmable Logic Arrays (Courtesy [7])

FPGAs are integrated circuits that contain large, two-dimensional arrays of homogeneous configurable logic blocks (CLBs). Each CLB has logical elements (LE). The logic element connects to a switch matrix to access the configurable distributed routing network. The LEs support both combinational logic and memory storage. Each FPGA logic element (LE) contains a look-up table (LUT) and a one-bit flip-flop (FF). This architecture has been widely adopted to speed up computationally intensive applications. As technology advances, modern FPGAs are getting denser and faster. A single FPGA chip has millions of gate equivalents and clock speeds above 100 MHz. As shown in Figure 1.1, most current FPGA devices employ an island-style fine grained architecture *[8]*, with additional fixed-function heterogeneous blocks such as embedded block RAMs, DSP functions such as multiply and accumulate and general purpose embedded processors.

1.1.1 Embedded Block RAMs:

Embedded RAMs in FPGAs provide large storage structures. While the capacity of a given block RAM is fixed, multiple block RAMs can be connected through the interconnection network to form larger capacity RAM storage. A key limitation of block RAMs is they have only two access ports allowing just two simultaneous reads or writes.

1.1.2 Multiply – Accumulate Blocks

The multiply-accumulate blocks, also referred to as DSP blocks, have dedicated circuitry for performing multiply and accumulate operations. These DSP blocks can also perform addition, subtraction and barrel shifter functions.

1.1.3 Embedded Microprocessor Blocks

The major FPGA companies provide embedded cores, both hard and soft, for use with their processors. Altera has the Nios II soft core [1] and Xilinx offers the MicroBlaze soft [2] and PowerPC hard cores [3] on their FPGAs.

All these large embedded logic blocks make more efficient use of on-chip FPGA resources. However, they can also waste on-chip resources if they are not being used. In this work, we will explore the utilization of these embedded blocks on Xilinx Virtex FPGAs in implementing floating-point operations and vector processing.

1.2 Fixed point versus Floating point arithmetic

Every numerical value is composed of an integer part, fractional part and the delimitation which is called the radix point. Two formats are popular for storing and manipulating data in the computational units, fixed point and floating point. In fixed point format, the radix point is always at a predetermined position while the position of the radix point is not fixed in floating point.

1.2.1 Fixed Point Data Format

radix point

0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	0	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0

Figure 1.2 Data representation in fixed point format

Several ways for representing values exist in the fixed point format: including signed fixed point and unsigned fixed point numbers. In the unsigned fixed point format, only positive values are represented. For 32-bit data, assuming the radix point to the right of the least significant bit (i.e., only integer values are represented), the range using the unsigned fixed point format is 0 to (2^{32} - 1). Assuming the radix point at left of the most significant bit (i.e., only fractional values are represented), the smallest representable fixed point value is 2^{-32}. In the signed fixed point format, the most significant bit is reserved for sign bit of the data and hence the range of the data can be -2^{31} to (2^{31} - 1). For the given data width, the range and precision of the fixed point value will vary based on the radix point's position. Figure 1.2 shows the numerical value 429.8515625 in unsigned fixed point format.

1.2.2 Floating point data format

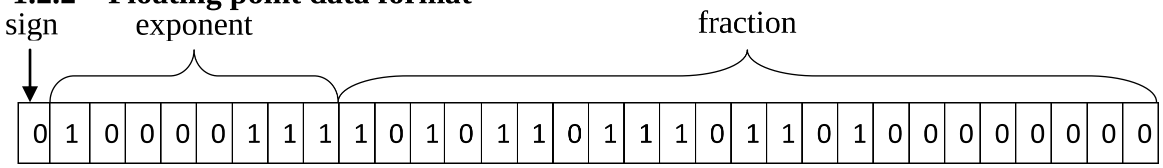


Figure 1.3 Data representation in Single precision floating point format

A value represented in floating point format is divided into three fields: the sign field s, the exponent field (e) and the fraction field (f). A floating point number can be defined as

$$(-1)^{s} * 1.f * 2^{e\text{-}BIAS}$$

The sign bit: 0 denotes a positive number and 1 denotes a negative number. The biased exponent is the sum of the exponent and a constant (bias value) chosen to make the biased exponent’s range nonnegative. The mantissa represents the magnitude of the number. When a number is normalized, the mantissa is composed of a leading one bit to the left of its implied binary point and the fraction bits to the right. In this representation a wider exponent field brings higher range, while a wider fraction field brings higher precision. The IEEE-754 standard [9] defines a single precision floating point number to have an exponent field 8 bits long and a fraction field 23 bits long, while for a double precision floating point number the exponent and fraction fields are 11 and 52 bits long respectively. These two IEEE floating point representations are the most commonly used in computer systems where fixed hardware architecture is necessary.

The main advantages of representing data in fixed point format are simplicity in the design and low cost [10]. But it suffers from low range and precision compared to floating point. Higher quantization error and increased algorithm complexity are two other issues that are inherent in fixed point arithmetic.

1.2.3 Floating Point Libraries

There are many libraries available to generate floating point operators for FPGAs. Lienhart et al. [11] have developed a library which includes adder, multiplier, divider and square root unit. A library unit supports generalized floating-point representation with generic exponent and mantissa bit. They have implemented these library components on Xilinx XC2V3000 FPGA to accelerate the smoothed particle hydrodynamics (SPH) algorithms for N-Body simulations. The parameterisable floating-point cores that first supported IEEE double-precision format [12] were implemented on Xilinx Virtex-II XC2V1000 and support addition, multiplication, division and square root operations.

Govindu et al. [13] have developed an IEEE-754 standard compliant double precision library. Instead of parameterizing range and precision of the computation units, they parameterized the pipelined depth and level of compliance with the IEEE-754 standard. All the floating-point computation units in this library were applied to the force and potential calculation kernels in a Molecular dynamics (MD) simulation.

Matousek et al. [14] used Logarithmic Number System (LNS) arithmetic instead of floating point arithmetic. Their library includes addition, multiplication, division, square and square root units and supports 32-bit and 20-bits precision. In the Logarithmic Number System multiplication, divide and square root operation are as simple as integer addition and subtraction whereas the main complexity lies in the addition and subtraction operations.

The Lyon Library [15] supports both floating-point and logarithmic number system arithmetic. This library is implemented in both pure combinational and pipelined versions. Apart from basic arithmetic unit, it also supports exponent and logarithmic conversion units.

VFLOAT library [16], which we used to generate single precision floating point unit supports adder, multiplier, divide, square root, format conversion units and accumulator. Wang [17] has demonstrated K-mean clustering and QR decomposition applications using these library operators. The key elements of the VFLOAT library include:

- Each component in the VFLOAT library has a READY and a DONE signal along with STALL signal to support pipelining. All the hardware modules in the library are fully pipelined.
- The VFLOAT library has a separate denormalization, normalization and rounding unit supporting either ”round to zero” or ”round to nearest”.
- The Library elements support some error handling and exception detection.
- The VFLOAT library has components to convert from fixed-point format to floating point format and vice versa, which supports the implementation of hybrid designs with both fixed-point and floating-point formats in a single design.

In summary, the VFLOAT library is a superset of many previously published floating point formats, and is general and flexible. Except for the Lyon library, the library is also more complete than earlier work with support for separate normalization, rounding and some error handling. Basic operational units such as adder, multiplier, divider and square root are generated from the parameterized floating point library or using CAD tools. In this research, we will focus on the implementation of a floating point unit with the help of the VFLOAT [16] library functions.

1.3 Floating point arithmetic unit on FPGA

The implementation of a floating point unit in general purpose computing is extremely common but it makes an interesting case study for an FPGA based reconfigurable computing system. Up to now there have been many research efforts applied to the implementation of an FPGA based Floating point unit. This research can be categorized based on the type of communication with main processor, precision support, number of computation units and level of autonomy.

One of the earliest works in this area is done by Fagin et al. [18]. They have implemented IEEE-754 standard compliant floating point adder and multiplier function on the FPGA for design space exploration. They found that the floating point unit substantially improves performance, but technology limitations made it difficult to implement floating point units at that time. Recently, Pittman et al. [19] have implemented a custom floating point unit (CFPU) which is compliant with the IEEE 754 standard and improves floating-point intensive computation performance. The CFPU is implemented on the Xilinx's Virtex FPGA based eMIPS platform which is partitioned into fixed and reconfigurable regions. They demonstrated various trade-offs for area, power and speed with the help of software profiling and run time reconfiguration for the CFPU.

The Altera's Nios II processor [1] allows a user to include floating-point custom instructions that implements single precision floating-point arithmetic operations. These custom instructions are used to accelerate floating-point operations in Nios II C/C++ application program. The basic set of floating-point custom instructions includes single precision floating-point addition, subtraction, and multiplication. Floating-point division is available as an extension to the basic instruction set. These are either implemented in software emulation mode or with hardware support.

The Xilinx’s MicroBlaze processor [2] supports a single precision floating point unit in hardware that is tightly coupled to the embedded processor pipeline. The FPU implements addition, subtraction, multiplication and comparison. Floating point division and square root are available as an extension, as well as conversions from integers to floating point and vice versa. If the FPU is not instantiated, floating point operations are emulated in software. The Xilinx’s PowerPC hardcore processor [3] has a floating point unit available that supports IEEE-754 floating-point arithmetic operations in single or double precision. Floating point instructions supported include add, subtract, multiply, divide, square root and fused multiply-add instructions. The FPU is tightly coupled to the PowerPC processor core with the Auxiliary Processing Unit (APU) interface [4]. The Xilinx FPU includes 32 floating point registers.

In other research, Govindu et al. [20] described and evaluated performance of dual-precision, pipelined, floating point arithmetic cores for addition/subtraction, multiplication and division. Each of the arithmetic cores can be switched at run-time to perform either one double-precision operation or two single-precision operation with the same hardware resources. Similarly, Schulte et al. [21] have presented an interval arithmetic [22] unit based variable precision floating point co-processor. The co-processor gives the programmer the ability to specify the precision of the computation, determine the accuracy of the result and recompute inaccurate results with higher precision. For similar accuracy in the result with comparable performance, the area requirement of their coprocessor is 20% lesser than the IEEE-754 standard double precision floating point unit.

Brunelli et al. [23] and Rodolfo et al. [24] have implemented flexible floating point units for open source MIPS based processors. These co-processors are not tightly coupled with the main processor and they also implement separate memory access unit within floating point unit. All floating point data accesses are done by these memory units which reduces traffic on the main processor bus and provides some degree of autonomy to the floating point cores. Instructions are still provided by the main processor.

The Ramp project from Berkeley has implemented RAMP Blue, an implementation of over one thousand cores implemented from MicroBlaze processors as a platform for manycore research. The authors connect the processors to an independent double precision floating point unit via FSL (Fast Simplex Link) [25]. Their main goal in implementing the independent FPU is to keep its pipeline full all the time. Others have implemented an autonomous FPU which can be shared by two processors [26]. Their FPU has separate register file, instruction fetch and write back unit. In their approach, loads and stores are handled by the processor which limits the amount of parallel execution that they can achieve. Recently, researchers have also proposed some limited hardware assist for the Altera SoftFloat library [27]. Their goal is to provide performance with a very small hardware cost. Each instruction is under control of the Altera Nios processor. This is very different from our goal to exploit the parallelism available on an FPGA.

As the available computational resources on a single FPGA increases, multiple floating point units can be implemented on the same FPGA. Researchers have published accelerating MicroBlaze floating point operations using an independent FPU [28]. The authors have implemented Single Instruction Multiple Data (SIMD) based vector floating point unit and suggested that, by integrating DMA and local memory within the floating point unit, they can reduce system bus traffic and improve the performance of the FPU. Their FPU only supports addition, subtraction and multiply operations while we support division and square root as well.

Yang et al. [29] have designed a floating point vector unit which is tightly coupled with the scalar processor. They have implemented floating point adder and multiplier in the floating point vector unit and used the vector unit to solve sparse matrices based on real flow network's linear equations. In similar research [30], Chen et al. proposed unified floating point unit approach to improve overall performance of the application. They conclude that by executing a separate vector thread on each lane can greatly improve the performance.

Only the last two researches are closely related to our research as both floating point unit implements Hybrid vector-SIMD computational model (refer section 2). But vector cores are still tightly coupled with the scalar core, which is responsible for integer arithmetic and program flow control. Also, in these floating point units, the number of floating point vector lanes and register file size are fixed and cannot be changed. In our research, we implement a completely autonomous floating point vector unit where the register file size and floating point vector lanes are configurable. Also, they have assumed that the targeted application does not perform any inter lane communication.

1.4 Contribution

The main contribution of this research is applying a unified scalar and vector processing model to data-parallel floating point applications. Such a soft vector processor provides a scalable and user-selectable amount of acceleration and resource usage, and a configurable feature set to the user. The scalability of the vector processor allows users to make large performance and resource tradeoffs in the vector processor with little or no modification to software.

We are also developing a software library that can adapt to the floating point unit's configuration. We include linear algebra kernels such as matrix multiplication, QR decomposition and Cholesky decomposition. Use of such a library makes accessible to the user a much larger design space and larger possible tradeoffs than current soft processor solutions. This architecture independent library allows acceleration of multiple sections of an application and multiple applications.

As part of this research, a complete vector processor was implemented within an embedded processor system. It targets a Xilinx Virtex V FPGA to illustrate the feasibility of the approach and possible performance gains. The co-processor instruction set architecture is highly inspired by the VIRAM instruction set architecture (ISA) [31], but makes modifications to include the scalar ISA features for FPGAs. A novel instruction execution model that is a hybrid between traditional vector and single-instruction-multiple-data (SIMD) is used in the co-processor.

1.5 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 gives an overview of vector processing and previously implemented vector processors. Chapter 3 describes in detail the architecture of the floating point vector co-processor. Chapter 4 provides software library description, experimental results and describes various design trade-offs. Chapter 5 summarizes the work in this thesis and provides suggestions for future work.

2 VECTOR PROCESSING OVERVIEW

Most current microprocessors have *scalar* instruction sets. A scalar instruction set is one which requires a separate opcode and related operand specifiers for every operation to be performed. Vector processors provide vector instructions in addition to scalar instructions. This chapter reviews vector processing in general, defines terms used in the rest of the thesis and lists recent trends in vector execution architecture.

2.1 Vector Operation

```
for (i=0; i < n; i++)
          Y[i] = A[i] * x + Y[i];
```

Figure 2.1 SAXPY/DAXPY computational kernel

The code in Figure 2.1 is called *SAXPY/DAXPY* loop which forms the inner loop of the Basic Linear Algebra Subprograms library [42]. For the above code, *A* and *Y* are vectors and *x* is a scalar value. Each iteration of the SAXPY/DAXPY loop, performs below six steps.

1. Load element from vector A
2. Load element from vector Y
3. Load scalar value x
4. Multiply A’s element with x
5. Add result of multiplication to element of Y
6. Store result back in vector Y

For a scalar processor, these operations will be performed in a loop. A Vector processor provides direct instruction set support for operations on whole vectors i.e., on multiple data elements rather than on a single scalar value. This vector instruction specifies operand vectors and a vector length, and an operation to be applied element-wise to these vector operands. Assuming the vector length is equal to the number of elements in each vector register then the SAXPY/DAXPY operation can be performed with just six instructions. Thus, vector operations reduce the dynamic instruction bandwidth requirement.

2.2 Vector Memory and Vector Register Architecture

There are two main classes of vector architecture: *Vector Memory Architecture* and *Vector Register Architecture*. In vector memory architecture such as the CDC STAR 100 [32], operands are read from memory and results of the operation performed on operands will be stored back in to memory. In vector register architecture such as Cray series [33], operands are read from vector registers and results of the operation performed on operands are stored back in the vector registers. Vector memory architecture has higher memory bandwidth requirement than vector register architecture.

2.3 Vector Length Control

A vector processor has a natural length determined by the number of elements in each vector register, which is called the *Maximum Vector Length (MVL)*. It is highly unlikely that a given program will have vector length that equals MVL. The size of all the vector operations for *SAXPY*/*DAXPY* depends on "n", which may not be known until run time. The solution is to create a *vector-length register* (VLR). The VLR controls the length of any vector operation. However, the value of VLR cannot be greater than the natural vector length of the processor.

```
m = n; i =0;
while (m > MVL){
        for (j = 0; j < MVL; j= j++)
              Y[i*MVL+j] = A[i*MVL+j] * x + Y[i*MVL+j];
        m = m – MVL;i++;}
for (j = 0;j < m; j++)
        Y[i*MVL+j] = A[i*MVL+j] * x + Y[i*MVL+j];
```

Figure 2.2 Strip mined SAXPY/DAXPY code

If the vector length is longer than the maximum length, a technique called strip mining is used. As shown in Figure 2.2, strip mining is the generation of code such that each vector operation is done for a size less than or equal to the MVL. In our vector processing, strip mining will divide vector execution into two parts. The first *for loop* will be vectorizable for length equals to MVL, whereas, the second *for loop* is vectorizable for length less than MVL.

2.4 Vector Lane

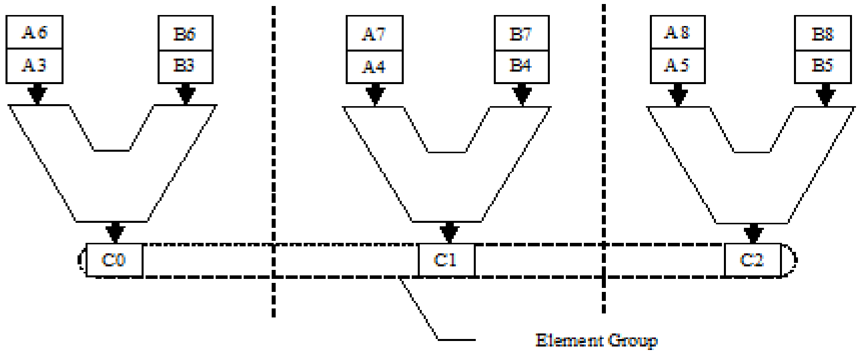


Figure 2.3 Vector lane diagram

The vector lanes of a vector unit are shown in detail in Figure 2.3. Each vector lane has a complete copy of the functional units, a partition of the vector register file and vector flag registers. All vector lanes receive the same control signals and operate independently without communication for most vector instructions. With more vector lanes, a fixed-length vector can be processed in fewer cycles, improving performance. Processor which supports vector operations through parallel lanes is generally known as Single Instruction Multiple Data (SIMD) processors. Many popular microprocessors have extended instruction set architecture (ISA) to support SIMD instructions such as Intel SSE, MMX and PowerPC AltiVec.

2.5 Vector Chaining versus Hybrid Vector-SIMD Model

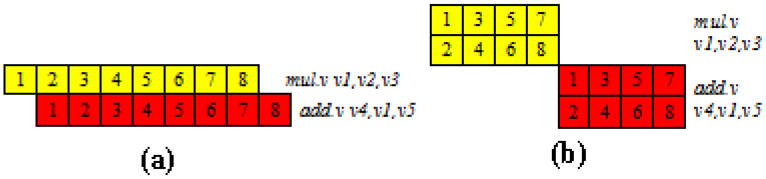


Figure 2.4 Vector chaining (a) versus hybrid vector/SIMD execution (b).

Traditional vector processors are optimized towards processing long vectors. Since they tend not to have a large number of parallel vector lanes, they rely on pipelining and instruction chaining through the vector register file to achieve high performance. Instruction chaining is illustrated in Figure 2.4(a). It refers to the passing of results from one functional unit to the next between two data dependent instructions before the entire result vector has been computed by the first unit. Chaining through the register file has a significant drawback: it requires one read port and one write port for each functional unit to support concurrent computations. This contributes to the complexity and size of the traditional vector register file. Another approach, which is described by Huang et al. [34] is to combine vector and SIMD processing and create a hybrid vector/SIMD model, illustrated in Figure 2.4(b). In the hybrid model, computations are performed both in parallel in SIMD fashion, and over time as in the traditional vector model. Elements which are accessed simultaneously for parallel execution contribute to short vectors whereas each vector register is made of multiple short vector registers. AMD's GPU architecture is one of the famous Hybrid vector/SIMD architecture.

2.6 Vector Memory Access Patterns

Below is the various memory access patterns, frequently used in vector processor. These patterns emerge from the sequence of vector memory instructions used to access an array operand within a vectorized loop. Our design supports all of these memory access patterns.

Strided Vector access: One of the simplest access patterns in which all vector elements reside in the memory at constant distance from each other and hence memory accesses are highly predictable. Most vector processors provide efficient strided access.

Permutations: A permutation access uses indexed vector memory instructions to access every item within memory and the index value is always constant. In this access, the index register is a scalar register.

Lookup Tables: In this type of access memory elements are accessed as per the index register and the index is a random value. The index register is a vector register. Such accesses can be observed in sparse matrix computations.

Neighbor Access: In this type of access, the same element in the memory is accessed multiple times by the neighboring lanes. This access can be imitated by providing stride value equal to zero.

Rake Access: This is a special two dimensional interleaved strided access and can be performed with a nested loop of vector memory instructions. This access is useful when you want to load or store multiple random rows or columns of a matrix.

2.7 Recent Trends in Vector Architecture

Many optimizations are implemented in vector processors. Recent trends in vector execution are listed below.

CODE (Clustered Organization for Decoupled Execution): A centralized vector register file limits the number of functional units because of its size and complexity. Kozyrakis [35] has designed a vector register file such that each vector functional unit can directly access only a subset of the registers. The issue logic selects the appropriate cluster to execute each instruction and the interconnection network between each subset of the register file provides results between vector functional units.

Decoupled Vector Execution: Espasa and Valero [36] have implemented partial dynamic scheduling by splitting the execution pipeline into three different streams: One implements the vector computation, the second contains all the memory access instructions and the third executes the scalar computation instructions. The decoupled pipelined architecture can implement limited chaining with minor increase in register file size and is able to support different latency for each vector stream.

Out-of-order Vector Execution: Register renaming and out-of-order issue in a vector processor improves performance. Performance improvements can be achieved by introducing extra physical registers for renaming. Renaming enables precise exceptions to be easily implemented [37].

Vector Lane Threading: Vector lane threading (VLT) allows short-vector or scalar threads to be run on idle vector lanes. The number of lanes assigned to each thread corresponds to its amount of data-level parallelism [38].

2.8 Conclusions

In this chapter we have given a brief introduction to vector processing, basic terminology used in the vector processor and recent trends in vector execution. In FPGA implementations usually a general purpose register file is implemented in block RAM which has only one pair of read write ports. Hence, it is hard to implement vector chaining on FPGAs. Our Hybrid vector/SIMD computation model can provide comparable performance. Also, we have implemented a unified vector scalar computation approach to save on area, and adopted decoupled vector execution to support variable pipeline latency for floating point operation. The details of our implementation are provided in the next chapter (Chapter 3).

3 FLOATING POINT VECTOR CO-PROCESSOR DESIGN

This chapter describes the design and implementation of the Floating Point Vector Co-processor (FPVC) targeting Xilinx FPGAs. Three key features distinguish our work in floating-point architecture: a unified approach to scalar and vector processing, support for different latency of each functional unit and simplicity of organization. The initial section of this chapter provides an overall architecture of the processor. Next section describes the Instruction Set Architecture (ISA) features whereas the unified vector core is described in following sections. Finally, FPGA specific novel inter-lane communication features, vector compression and expansion and configurable parameters are described.

3.1 Floating Point Vector Co-processor Architecture

The Floating Point Vector Co-processor (FPVC) is a configurable soft-core vector processor architecture developed specifically for FPGAs. It leverages the configurability of FPGAs to provide many parameters to configure the processor for a specific application for desired performance and area. The FPVC instruction set supports both scalar and vector instructions with unified register files and execution units. Instruction set features are heavily borrowed from the instruction set of VIRAM and RISC processors such as PowerPC and Microblaze. Currently the FPVC does not support virtual memory and certain bit manipulation instructions, but it adds new instructions to take advantage of DSP functionality and embedded memories.

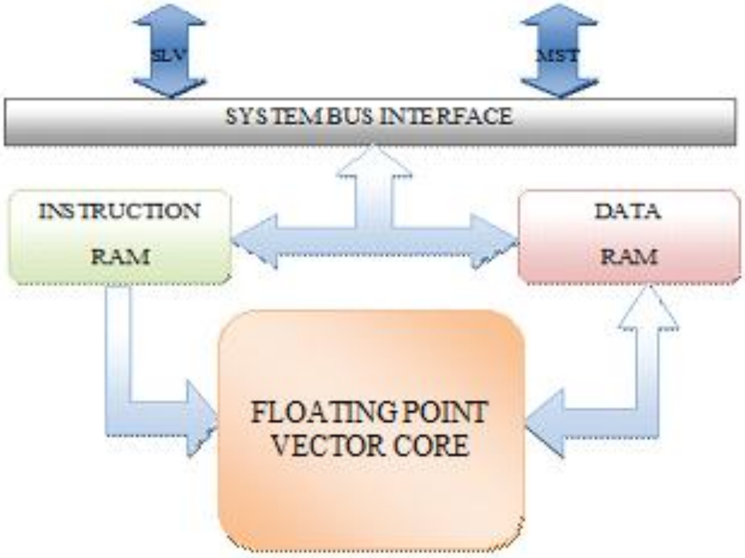


Figure 3.1 Vector Co-processor Block Diagram

Figure 3.1 illustrates the high level view of the floating point vector processor. The FPVC integrates local instruction and data RAM, system bus interface and a floating point vector core.

3.1.1 FPVC – The Vector Co-processor

Some of the key design features of Floating Point Vector Core are:

- Completely autonomous from the main processor
- Supports single precision and 32-bit integer arithmetic operations
- 4 stage RISC pipeline for integer arithmetic and memory access
- Variable length RISC pipeline for floating point arithmetic
- Unified vector scalar general purpose register file
- Supports modified Harvard style memory architecture where there are separate level 1 instruction and data RAM but unified level 2 memory

3.1.2 Local Instruction and Data RAM

The FPVC implements a modified Harvard style memory system architecture. The FPVC's memory system is divided in two levels: main memory and local memory. Main memory is connected to FPVC through master port of system bus interface whereas local memory sits in between. Apart from our approach, there are many options exist for connecting the FPVC to the main memory, such as through unified cache memory, separate instruction and data cache, through direct connection to main memory etc. For off-chip memory, caches are used to hide the memory latencies, but for streaming applications in the area of embedded and scientific applications this may not be true. This range of different memory system configurations could be interesting to explore in the future.

As shown in Figure 3.1 instruction and data RAM constitute local memory. These memories can be configured for various sizes with various data widths using *C_INSTR_MEM_SIZE, C_DATA_MEM_SIZE and C_MPLB_DWIDTH* parameters at design compile time. The instruction and data memory are part of system address space and can be accessed by any master on the system bus. Hence, data coherency and other communication protocols between FPVC and other masters can be implemented through software. Also through software, we can pre-fetch instructions and data required for FPVC and reduce traffic on the system bus.

3.1.3 System Bus Interface

The FPVC has one slave (SLV) port for communicating with the main processor and one master (MST) port for main memory accesses. The system bus interface for master and slave ports is not restricted to a specific bus protocol. The slave port interface can be connected to any type of bus including point-to-point, shared bus or simple glue logic.

In the current design implementation, we have implemented Processor Local Bus (PLB) [3] as the system bus interface. The PLB can be configured for 32-bit, 64-bit or 128-bit interface. Data alignment is also done in the system bus interface. The master port of the system bus interface includes a Direct Memory Access (DMA) controller to provide software prefetch mechanism for the vector core. DMA transfers setup include below three steps.

1. Write source/destination address of main memory in global address register
2. Write destination/source address of local memory in local address register.
3. Configure DMA transfer parameters : includes direction of the DMA, size of the DMA, which local memory involves in the transfer, state of DMA transfer are provided written in configuration register

Details for each register are provided below.

1) Control Register (CR) :-

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FPU EN	INT EN	S/W RST	RSVD	DMA EN	DMA WAIT	DMA TYPE	DMA RNW	RSVD							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RSVD				DMA LENGTH											

FPUEN (31) : Enables or disables the floating point co-processor (0 - Disable, 1 - Enable)

INTEN (30) : Interrupt enable (0 – Disable, 1 – Enable)

S/W RST (29) : Software reset (0 - No effect on co-processor, 1 – Reset co-processor)

DMAEN (27) : Enables DMA transfers (0 – Enable DMA, 1 – Disable DMA)

DMAWAIT (26) : Core execution state when DMA transfer is enabled (0 – Continue, 1 – Pause)

DMA TYPE (25) : Destination or source of DMA (0 – Instruction RAM, 1- Data RAM)

DMA RNW (24) : Direction of DMA (0 – DMA write operation, 1 – DMA read operation)

DMA LENGTH (11-0) : Length of DMA transfers in number of bytes

2) Global address register (GAR):-

It provides destination/source address to DMA controller for main memory.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Global Address															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Global Address															

3) Local address register(LAR) :-

It provides destination/source address to DMA controller for local memory.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Local Address															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Local Address															

3.2 Vector-Scalar ISA

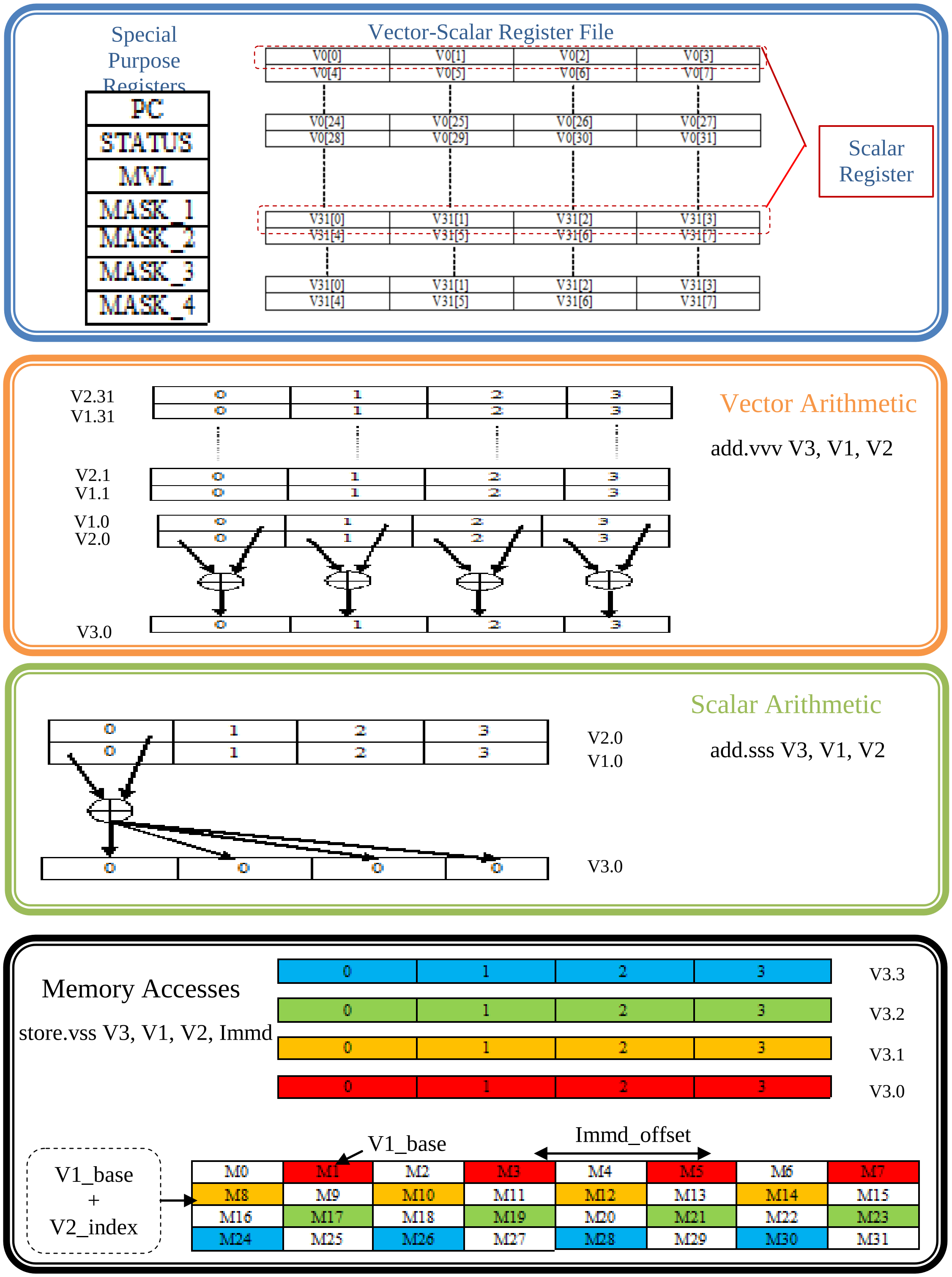


Figure 3.2 User programming model of a vector scalar architecture

This programming model has 32 vector data registers, within each register there are 32 short vectors and each short vector register has 4 lanes which totals 128 elements in a single vector register.

The goals of the Vector-Scalar ISA design are to be flexible and scalable and to provide a simple architecture suitable for a wide range of floating point applications. Due to trade-offs between performance versus design complexity, we selected to design new ISA which is inspired by RISC instruction architectures such as Power ISA [3], Microblaze ISA [2] and VIRAM vector ISA [35].

The Vector-Scalar ISA is a 32-bit instruction set. The instruction encoding allows for 32 vector-scalar registers with variable vector length. As shown in figure 3.2, the top short vector of each vector register can be used as a scalar register. Due to this unification of the vector register file we can freely mix vector and scalar vector registers without any scalar – vector core communication. Register-0 always returns the value zero. Each vector register supports configurable lane width. The vector-scalar ISA supports maximum vector length of *C_NUM_OF_LANE* * *C_NUM_OF_VECTOR.* These parameters are explained in section 3.5. Vector-scalar instruction can be classified into three major classes:

1. Memroy access Instructions
2. Artihmetic Instructions
3. Inter lane communication instructions

This classes are described below whereas each instructions are defined in Appendix A.

3.2.1 Memory Access Instructions

Vector memory instructions (load.xxx and store.xxx) support all memory access patterns described in Chapter 2. Figure 3.2 bottom shows the rake access pattern, which can be described by a single vector instruction in the vector scalar ISA. Rake access is generally used in block operations for matrices. It requires a vector base address, vector index address and immediate offset value, which is a distance between two neighbor elements. All neighbor elements within each rake are stored in single short vector whereas each rack of elements is stored in different short vector. The same instruction can be used for unit stride and non unit stride access by setting the immediate offset value to an equal distance between two vector elements in memory. Permutation and look-up table access classes are realized by keeping immediate offset to zero and providing index register. The only difference between permutation and look-up table accesses is the index register is scalar and vector type respectively. Vector ISA designs anticipate that most hardware would be optimized for unit-stride or indexed accesses; in our vector-scalar architecture all memory accesses are performed equally efficiently.

As source and destination registers are encoded in the same opcode space, scalar accesses are supported with the same instruction. It also allows a post-increment of the scalar base address to be specified in the same instruction for indexed and rake accesses. This post-increment is an arbitrary amount taken from any scalar register, and register zero can be specified if no post-increment is required. This post-increment avoids a separate scalar add instruction, and so saves instruction issue bandwidth.

3.2.2 Arithmetic Instruction and Instruction masking

The primary application domain of our research is floating point arithmetic applications and hence we have included several floating point instructions: floating point add, multiply, divide and square root. In order to be an autonomous and to support basic program control constructs, we have included basic integer arithmetic, compare and shift instructions which operates on the full data word. One exception is the multiply instruction which only works on the lower 16-bits of the operands and produces a 32 bit result.

In conventional vector ISA or VIRAM ISA based vector processors [35] [39] [5] [6], that implement separate scalar and vector processor cores, inter processor communication instructions are usually provided to allow the scalar processor to access single elements within a vector register. These are useful for partially vectorizable loops. A common example is where a loop contains memory accesses that can be vectorized but where the computation contains a dependency between loop iterations that requires scalar execution. In our Vector-Scalar ISA, all scalar operations are performed on the first element of the first short vector of each register and the result will be replicated to all lanes and stored on the first short vector of the destination register. Hence, vector instructions which require scalar data can reference the top of each register. Figure 3.2 shows vector and scalar arithmetic operation.

Masked vector instruction execution is usually provided to allow vectorization of loops containing conditionally executed statements. A mask vector controls the element positions where a vector instruction is allowed to update the result vector. The mask vector may be held in special flag/mask registers.

3.2.3 Inter Lane Communication Instruction

Each lane in the vector processor communicates with each other via local memory access instructions which access local data memory. In some cases, other forms of inter-lane communications can improve performance but these can incur significant hardware costs to provide the necessary inter-lane communication. These Inter-lane communication operations are vector compression and expansion. Compress instructions select the subset of an input vector marked by a flag vector and pack these together into contiguous elements at the start of a destination vector. Expand instructions perform the reverse operation to unpack a vector, placing source elements into a destination vector at locations marked by bits in a flag register.

Compress and expand operations can be used to implement conditional operations in density-time. Instead of using masked instructions, flagged elements can be compressed into a new vector and vector length reduced to lower execution time. After the conditional instructions are completed, an expand operation can update the original vectors. But, as the number of lanes increases and the number of clocks per vector instruction drops, the advantages of compress/expand over masked instruction execution diminishes.

There are several alternative ways of providing compress and expand functionality. In traditional vector computers, vector compression and expansion is implemented using scatter and gather instructions. In this approach, the masked bit is applied to the index vector and compressed indices are written to the index vector. The elements can then be retrieved using a vector gather instruction. Another variant of this approach is to support compress and expand functionality as part of loads and stores. The loads compress flagged elements as they were read from memory, and compressed stores compress elements from a vector register storing the packed vector in memory. Expand variants can also be supported in the opposite manner. For cases where the packed vector must move between registers and memory, these instructions avoid the inter-lane traffic and excess memory traffic required for register-register compression or compressed index vectors. Adding these compressed memory instructions does not require much additional complexity on top of masked load/stores, but it does require additional instruction encodings.

T0 [39], VIRAM [38], VESPA [5] and soft vector accelerator [6] use the crossbar wiring to perform inter-lane communication. In this approach flagged registers are only routed to the destination register. This register-register compress avoids the use of memory address bandwidth to move data and so may have the highest performance for machines with limited address bandwidth at the expense of higher design complexity of the load store unit. The index vector compression is easy to simulate with a register-register compress.

We have implemented a register-register instruction that reads a source flag register and a source vector data register, stores it to a FIFO inside the load store unit and writes these flagged data into the first elements of the destination vector register in serial fashion. The advantage of this approach is that it does not require access to memory and it does not require a crossbar in the load store unit. Expand instructions are implemented similarly.

3.3 Vector-Scalar Pipeline

The FPVC is based on the classic dynamic scheduling (In-order issue and our-of-order completion) RISC pipeline. The four stages of the pipeline are Instruction Fetch, Decode, Execution and Write Back. The pipeline is intentionally kept short so integer vector instructions can complete in a small number of cycles to eliminate the need for forwarding multiplexers and to reduce area. Due to the short pipeline, floating point instruction spends most of their time in the floating point unit which optimizes the overall execution latency. As both scalar and vector instructions are executed from the same instruction pipeline, both type of instructions are freely mixed in the program execution and stored in the same local instruction memory.

As shown Figure 3.3, the fetch and decode stages are common to all instructions. The instruction fetch stage (IF) is used to access the instruction. During IF stage, the next instruction address is determined and sent to local instruction RAM, which returns the instruction by the end of the cycle. We assume that all instructions fit in the local instruction RAM. Therefore the size put a limitation on the size of program. An alternative approach is an instruction cache which would support larger programs at the cost of increased complexity. The fetch unit of the pipeline always assumes that branches are not taken and fetches an instruction from the next instruction address. Hence, the branch penalty for the vector co-processor will be two cycles.

During the decode stage (ID), instructions pass through three steps and all steps are performed in a single clock cycle. In the first step, the instruction is decoded and data hazard checkers perform checks to find out whether the current instruction's source and/or destination registers are in flight. Due to dynamic scheduling, not only RAW

(Read after Write) hazards exist but the data hazard checker must also check WAW (Write after Write) and WAR (Write after Read) hazards. Until all hazards are cleared, the Instruction Decode unit stalls the next instruction fetch.

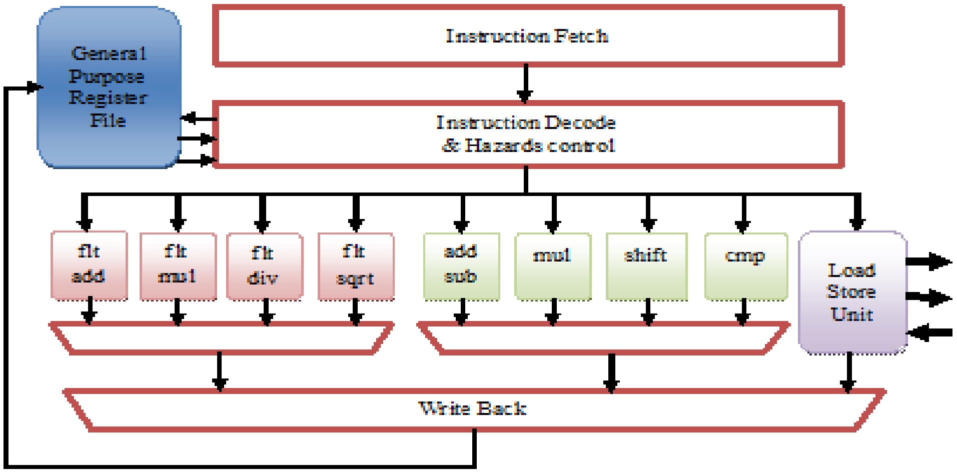


Figure 3.3 Vector Co-processor Pipeline

Once all hazards are cleared, the instruction enters the vector state machine. If the instruction is a scalar instruction then the vector state machine issues the instruction to the execution unit and the new fetched instruction enters the decode stage in next clock cycle. If it is a vector instruction, based on the maximum vector length stored in the MVL register, the instruction will be repeatedly issued to the execution unit.

As the instruction encoding can address 32 short vector registers as source/destination registers, the vector counter together with source/destination register references the current instruction’s operand data. Finally, in the last step, when the decode stage issues an instruction to the execution unit, it updates the scoreboard to keep track of in flight instructions for data hazard checks.

Once the instruction reaches the end of the execution unit, the result is written back to registers.

3.3.1 Execution Unit

The execution unit performs floating point arithmetic (add, sub, mul, div, sqrt), integer arithmetic (add, sub, mul), shift operations, comparisons of the operands and memory accesses to local data memory. These functions are shown in Figure 3.3. The execution unit is divided into three main units based on the type of data and memory access for decouple execution. These three units are floating point unit (FPU), integer unit (IU) and load store unit (LSU) and all these units work independently from each other. To keep our overall design simple, instruction decoding is divided into two parts. First, partial decoding is done in the decode stage which is useful for data hazard checking and scoreboarding. The second part is implemented in the execution unit and directs the current instruction to a particular unit.

MODULE	**Latency**
Denormalization	0
Normalization	1
Rounding	1
Add	4
Multiply	4
Division	10
Square root	11

Table 3.1 Floating Point Operations and their Latencies

Normalization and rounding for the floating point arithmetic functions are done along with multiplexing in the data path. The IEEE 754 standard single precision functional units are generated from the VFLOAT library [16]. Latency for each function unit is shown in Table 1.1.

Modern FPGAs include fixed computational units such as adders, multipliers and barrel shifter as well as storage elements such as block RAMs. All computational units are pipelined so they can produce result in single cycle with a reasonable operating frequency.

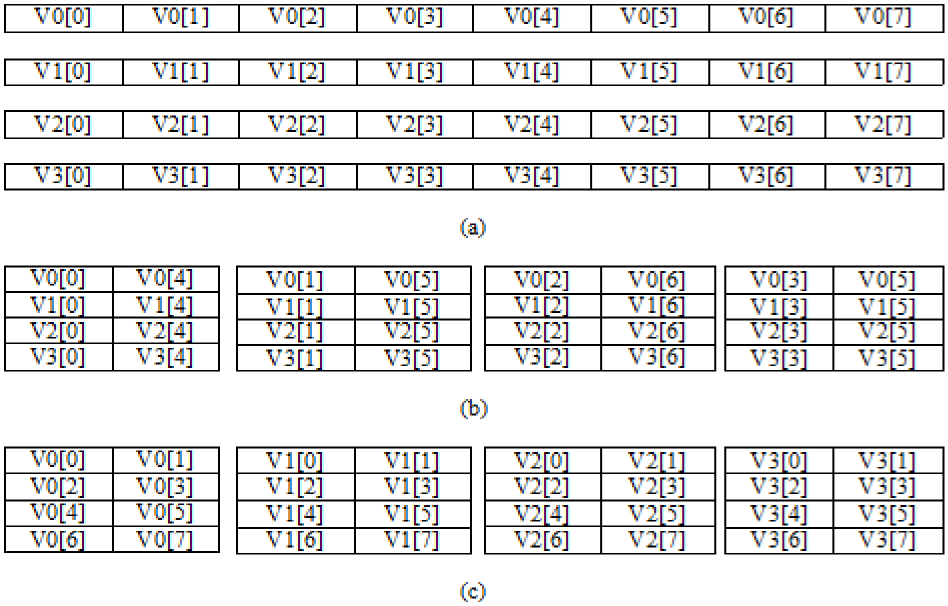
Since the execution unit supports dynamic scheduling all three major units works independently from each other. Hence, a structural hazard may occur at the end of execution of an operation. To avoid this problem, we have implemented an arbiter between the end of the execution stage and the write back stage of the pipeline. This arbiter can commit one result in each clock cycle. When multiple results are available at the same time, one will be written to the register file, and other results will be stalled. We currently implement a straightforward fixed arbitration scheme with the load store unit having the highest priority and integer arithmetic with lowest priority, but will consider other schemes in the future.

3.3.2 Vector Register File

The vector register file lies at the heart of the vector core. It provides temporary storage for intermediate values and interconnection between functional units. The number of registers and length of each vector register is a key decision in the vector core design. Here, first we will discuss various ways to configure vector registers and configurations in previous vector processors and then we present our implementation of the vector register.

Register-Partitioned versus Element-Partitioned Register Banks

There are two orthogonal ways in which we can divide the vector register storage into banks. The first places elements belonging to the same vector register into the same bank, which is called *register partitioned*. The second places elements with the same element index into the same bank, which is called *element partitioned*. A separate interconnection network connects banks and function unit ports.



V0[0]	V0[1]	V0[2]	V0[3]	V0[4]	V0[5]	V0[6]	V0[7]
V1[0]	V1[1]	V1[2]	V1[3]	V1[4]	V1[5]	V1[6]	V1[7]
V2[0]	V2[1]	V2[2]	V2[3]	V2[4]	V2[5]	V2[6]	V2[7]
V3[0]	V3[1]	V3[2]	V3[3]	V3[4]	V3[5]	V3[6]	V3[7]

(a)

V0[0]	V0[4]	V0[1]	V0[5]	V0[2]	V0[6]	V0[3]	V0[5]
V1[0]	V1[4]	V1[1]	V1[5]	V1[2]	V1[6]	V1[3]	V1[5]
V2[0]	V2[4]	V2[1]	V2[5]	V2[2]	V2[6]	V2[3]	V2[5]
V3[0]	V3[4]	V3[1]	V3[5]	V3[2]	V3[6]	V3[3]	V3[5]

(b)

V0[0]	V0[1]	V1[0]	V1[1]	V2[0]	V2[1]	V3[0]	V3[1]
V0[2]	V0[3]	V1[2]	V1[3]	V2[2]	V2[3]	V3[2]	V3[3]
V0[4]	V0[5]	V1[4]	V1[5]	V2[4]	V2[5]	V3[4]	V3[5]
V0[6]	V0[7]	V1[6]	V1[7]	V2[6]	V2[7]	V3[6]	V3[7]

(c)

Figure 3.4 Vector register storage partitining

Register partitioning is created for an example vector register file with 4 vector registers each 8 elements long. All vector registers are addressed in Vx[y] format, where x denotes register number, y denotes the element number. All configurations have been divided into 4 memory banks.

Figure 3.4(a) shows a vector storage scheme which is popular in traditional vector processors and SIMDized vector processor. This is the simplest vector storage scheme as it does not require any interconnection network between banks and functional units. With this scheme, a functional unit reserves a bank port for the duration of an instruction's execution. If another functional unit tries to access the same bank using the same port, it must wait for the first instruction to finish execution.

Traditional vector processors are heavily dependent on long vectors and vector chaining, but because of the exclusivity of each instruction on a single bank port, chaining through the register file requires extra ports. Similarly, a SIMDized processor can only be scaled by the number of lanes but as you scale the lanes, resource usage and efficiency of resource utilization may be severely hit. Hence, this is a limited scaling option, even though it is the simplest to implement.

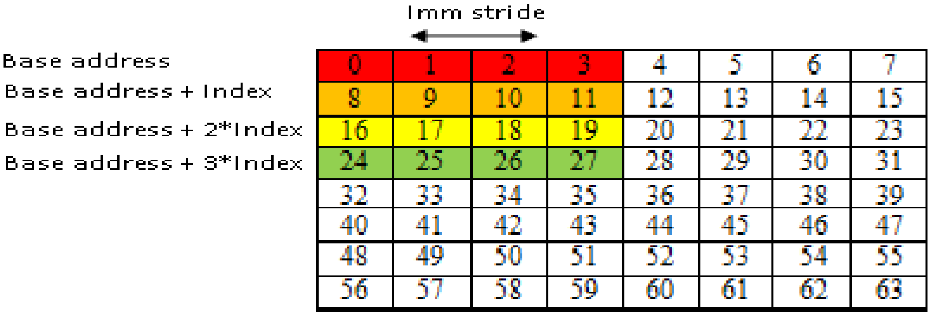
An element-partitioned scheme is shown in figure 3.4(b). By interleaving vector register storage across multiple banks, a single instruction cycles through all the banks one after another and hence, we can reduce the number of ports required on each bank. This allows all functional units to perform multiple chained accesses to the same vector register even though each bank has a limited number of ports. This scheme is implemented in the soft vector accelerator, VIRAM [35], T0 [39] and RSVP [40]. This can scale better than the register-partitioned scheme, but as the number of element increases so as the number of banks and the interconnection network will become bottleneck of the design.

In an FPGA, the vector register file is implemented in block RAM and this RAM has only one pair of asynchronous read and write ports, which lends itself to the element-partitioned scheme. But with the adaption of the Hybrid-SIMD model, we can partitioned long vectors into short vectors and are able to store the short vectors of each vector register in a single bank. Thus we can avoid the limitations of element-partitioned banks and achieve the simplicity of the register-partitioned bank. Hence we used a register storage scheme as shown in figure 3.4(c).

3.3.3 Load Store Unit

Memory instructions are dispatched to the vector load store unit controller at the end of the ID stage. The Load Store Unit contains a small memory controller which is a state machine that generates basic memory read-write signals and the pipeline control signals for the remainder of the memory pipeline. Basic stages of the state machine are memory address calculation, memory access and write back results to the vector register file. In the address calculation stage, the base register is updated with the index value based on access type. The load store unit supports vector unit stride, vector non-unit stride, index accesses and scalar access.

In traditional vector processors, multiple memory access ports or wide read-write ports are implemented. Multiple ports require that memory has multiple read-write port available whereas wide read-write ports only optimize unit stride operations. In an FPGA, the load store unit is connected to only on-chip block RAM memory, which has only one read-write port available. Hence, to support multiple memory accesses requires a crossbar and data alignment logic. To keep the design simple and to follow strictly sequential memory consistency model, we serialized all the memory accesses. Thus each memory access is performed in 3 clock cycles. One optimization is that the load store unit can do burst accesses for each short vector. Burst size depends on the number of lane parameter. So, if the vector scalar architecture has 32 short vectors per vector-scalar register and each short vector accommodates 4 lanes then the total number of memory access will be 128. These 128 memory accesses will be partitioned into 32, 4-beat bursts accesses.



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Figure 3.5 Matrix subset access

Two dimensional matrix accesses or rake accesses can be done with a single vector instruction. Here, we provided base address, arbitrary index value for post increment and immediate stride values.

3.4 Configurable Parameters

Parameter Name	Description	Typical Value
C_EXP_BITS	Exponent width for floating point data	8-11 bits
C_MAN_BITS	Mantissa width for floating point data	23-52 bits
C_ADD_MAN_BITS	Mantissa width for floating point addition	23-52 bits
C_MUL_MAN_BITS	Mantissa width for floating point multiply	23-52 bits
C_DIV_MAN_BITS	Mantissa width for floating point division	23-52 bits
C_SQRT_MAN_BITS	Mantissa width for floating point square root	23-52 bits
C_NUM_OF_LANE	Number of parallel functional units	1,2,4,8
C_VECTOR_LENGTH	Number of short vectors	1,2,4,8,16,32
C_INSTR_MEM_SIZE	Size of local instruction RAM	1K-64KB
C_DATA_MEM_SIZE	Size of local data RAM	1K-64KB

Table 3.2 Configuration Parameters

Table 3.2 lists the configurable parameters and features of the FPVC. As unified vector scalar register stores integer and floating point data, *C_EXP_BITS*, *C_MAN_BITS* will decide element size of the FPVC. Maximum Vector Length is determined by *C_NUM_OF_LANE* and *C_VECTOR_LENGTH*. Element size, the number of registers defined in the ISA and the FPGA embedded memory block size constrains Maximum Vector Length. *C_INSTR_MEM_SIZE* and *C_DATA_MEM_SIZE* determine the size of local memory. These are the primary parameters of FPVC that impact design. *C_ADD_MAN_BITS,C_MUL_MAN_BITS,C_DIV_MAN_BITS* and *C_SQRT_MAN_BITS* are the secondary parameters. We are using the VFLOAT [16] library design modules which support customized range and precision of each floating point unit. The secondary parameters enables us to do more fine tuning of the design according to the application requirements but currently we support only single precision floating point operations using these units.

3.5 Conclusion

In this chapter, we have presented the overall architecture of the floating point vector co-processor and proposed the new vector scalar ISA. We have discussed various design trade-offs in the co-processor pipeline and inter-lane communication operation, compression and expansion. Finally, we have listed various configurable parameters which can be defined for each application domain. True reconfigurable processor based system, not only provide configurable hardware but it should have adaptable software for different hardware configuration. In the next chapter, we present adaptable software design and results for different FPVC configurations.

4 RESULTS AND DISCUSSION

Numerical linear algebra kernels are key components in scientific computing, multimedia and human-machine interface tasks. With the rapid advances in technology, hardware acceleration of linear algebra applications using FPGAs has become feasible. In this chapter, we describe software kernels based on BLAS [42] and LINPACK [41] libraries that run on our vector scalar co-processor architecture. The design is implemented on the Xilinx's Virtex 5 FPGA and performance of each FPVC configuration for these kernels is compared against performance of an embedded processor block on Xilinx's Virtex 5 FPGA.

4.1 Experimental Setup

We have tested the FPVC for correctness as well as for area usage and speed. The FPVC is implemented in VHDL and synthesized using Xilinx ISE 10.1 CAD tools targeting Virtex-5 FPGAs. We have compared various FPVC configurations against a hard processor (PowerPC 440 with the Xilinx FPU [3]) using various linear algebra kernels. We have also compared area and speed for each configuration of the FPVC.

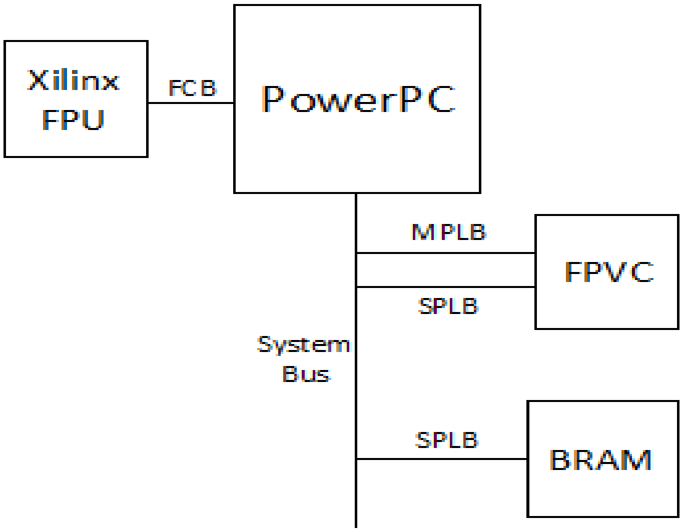


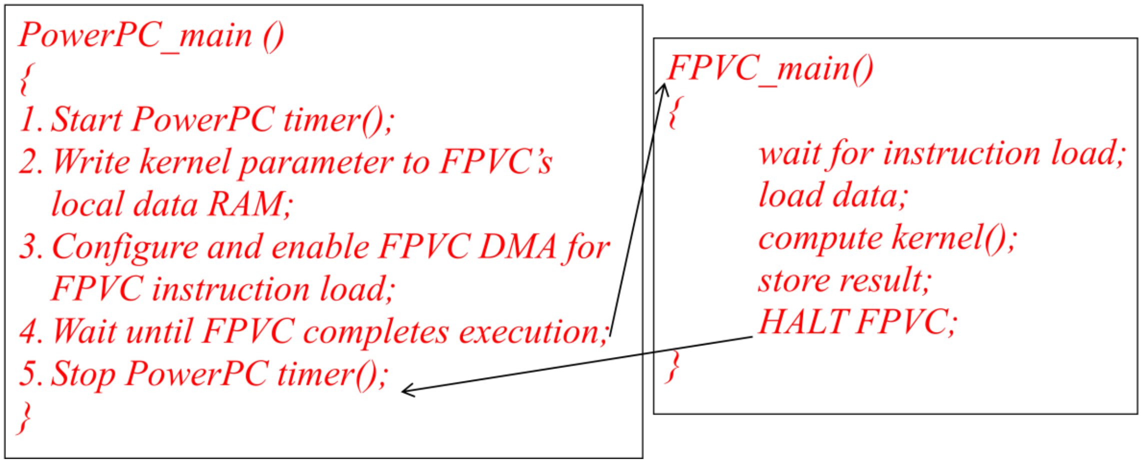
Figure 4.1 Embedded System with FPVC and Xilinx FPU

The Xilinx ML510 board [43] was used to test the design and both PowerPC's floating point extension Xilinx FPU [4] and the FPVC are implemented in a single embedded system as shown in figure 4.1. Xilinx FPU is directly connected to PowerPC using point-to-point FCB (Fabric Co-processor Bus) [44] whereas FPVC is connected through 32-bit Processor Local Bus (PLB) [4]. Embedded system is running at 100MHz.

Program Flow

PowerPC based linear algebra kernels are written in C and compiled using gcc with –o2 optimization. FPVC based linear algebra kernels are written directly in machine code, stored as data array in embedded system and no optimization is performed. Program and data are stored in on-chip BRAM (64 KB main memory, as shown in figure

4.1) of the embedded processor system. As all programs and data are stored in the main memory and FPVC can only access local memory (64 KB instruction and data RAM) (see section 3.1.2), FPVC system bus interface is responsible for communication between local and main memory.



```
PowerPC_main ()
{
1. Start PowerPC timer();
2. Write kernel parameter to FPVC's
   local data RAM;
3. Configure and enable FPVC DMA for
   FPVC instruction load;
4. Wait until FPVC completes execution;
5. Stop PowerPC timer();
}
```

```
FPVC_main()
{
        wait for instruction load;
        load data;
        compute kernel();
        store result;
        HALT FPVC;
}
```

Figure 4.2 Typical Program Flow for FPVC Kernel

Figure 4.2 shows typical program flow for PowerPC and FPVC. All the parameters required to execute the kernel on FPVC such as source addresses, destination addresses, size of the data etc., are written to the local memory by PowerPC. FPVC's DMA will be configured next. DMA will be configured using system bus interface's configuration register (refer section 3.1.3). Once, all instructions are loaded in the local instruction memory FPVC starts execution. While FPVC executes the kernel, PowerPC polls "DONE" bit of configuration register (which will be set by executing HALT instruction on FP vector core). The main performance metric is the number of clock cycles between the start and the end of a kernel. Clock cycles are counted using the PowerPC's internal timer and all results are normalized to PowerPC's runtime.

4.2 Linear Algebra

Numerical linear algebra, particularly the solution of linear system of equations, linear least square problems, eigenvalue problems and singular value problems, is fundamental to most embedded and scientific applications. As it is often the most computationally intensive part of such applications, the performance improvement of numerical linear algebra has always been of interest to researchers. Certain basic vector and matrix operations of the arithmetic libraries such as BLAS [42], LINPACK [41] have been implemented on FPGA. In [45] author has implemented basic BLAS operations on FPGA. Wang [46] has implemented QR decomposition in two-dimensional systolic array architecture on FPGA where as Maslennikow et al. [47] has implemented Cholesky Decomposition based least square problem on FPGAs. All of these researches have designed custom application specific floating point compute path and it cannot be used for other operations.

4.3 DOT Product

BLAS level 1 routines are in the form of 1D vector operations, with *O (N)* operations performed on length *N* vectors. One important routine is the vector dot product, which can be formulated as:

$$z = \sum_{i=0}^{N-1} u_i \, v_i$$

PowerPC writes source and destination address and vector size parameters on FPVC's local data RAM. Complete program flow is explained in section 4.1. Here we will only discuss compute kernel executed on the FPVC.

```
DOT_product_kernel()
{
  load vector u from local data RAM;
  load vector v from local data RAM;
  mul_vector = multiply u and v;
  accumulate = reduction(mul_vector);
  store accumulate to local memory;
}
```

Figure 4.3 Pseudo Code For DOT PRODUCT

Figure 4.3 shows pseudo code for DOT product. Strip mining (see section 2.3) is inherent in each kernel to handle any length of vector. Here the memory load instruction requires immediate value equals to constant stride value, which is a distance between two elements within vector whereas index value is equals to constant stride value times number of lane. In this way each short vector contains first MVL elements. Now that we have loaded both vectors, we can multiply each short vector's lane in parallel where as each short vector in every clock cycle and results are stored in general purpose register.

The reduction function accumulates all the elements in general purpose register using compress and expand instruction. The fundamental concept of reduction operation is to copy lower half of mul_vector to another register, add both the register and half the MVL value. This pattern is performed continuously until final scalar value will be remaining in the mul_vector. The rest of the DOT product code is self explainable. Results for various configuration of FPVC are given in the below figures and all the performance number are normalized with respect to PowerPC.

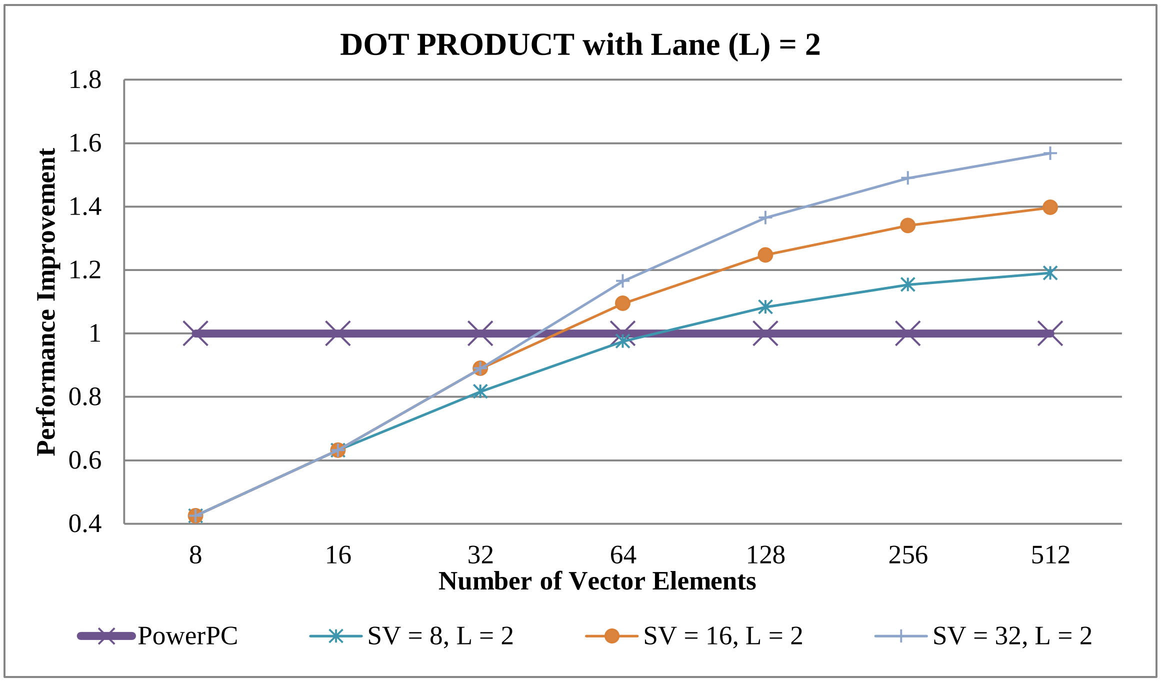


Figure 4.5 DOT PRODUCT Performance for short vector scaling

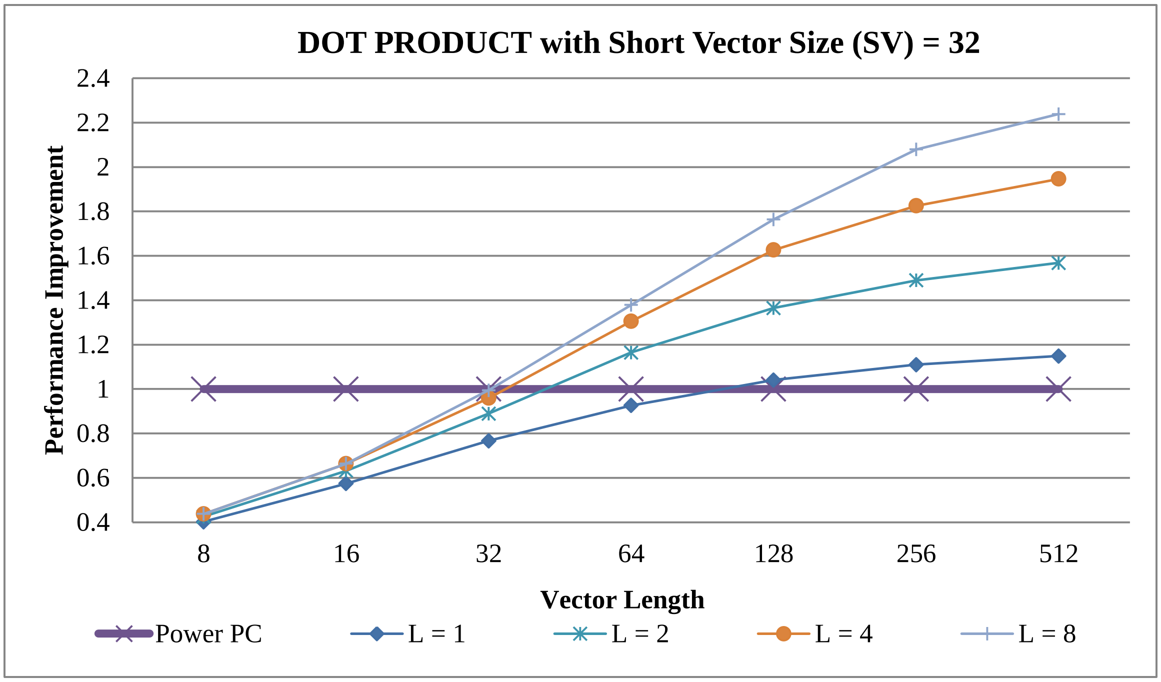


Figure 4.4 DOT Product Performance for Lane Scaling

ANALYSIS: Figure 4.4 shows performance for DOT product with short vector size (SV) = 8, 16, 32 and constant number of lane (L) = 2 whereas Figure 4.4, shows performance for DOT product with constant vector size. Some observations are listed below:

- As the FPVC requires all the code and data to be resided into local memory, (2N+1+no. of instructions) memory accesses will be performed to load local memory and to store result back to main memory. The significance of main memory access can be observed in both the graphs where the FPVC underperformed compared to the PowerPC.

- For vector length of N, N floating-point multiplication as well as N floating-point addition (total 2N floating point operations) needs to be performed and as number of lanes scales up the performance increases, which we can see in figure 4.5. Highest performance improvement is 2.24x over PowerPC for 512 elements for FPVC configuration, SV = 32 and L = 8.

- If complete vector is accommodated in single vector register no strip mining is required and hence the kernel execution time will be equal. For the FPVC configuration (SV = 8 and L =2; SV = 16 and L = 2; SV = 32 and L = 2) DOT product for vector size 8 and 16 are accommodated in single vector and hence execution time for these vector sizes is same. Similarly for FPVC configuration (SV = 16 and L = 2; SV = 32 and L = 2) performs 32 element vector DOT product in same execution time. But FPVC configuration (SV = 8 and L = 2) requires more execution time since all 32 elements are not accommodated in a single vector register and strip mining is required. The number of vector elements accommodated in single vector register is controlled by maximum vector length (MVL), which is controlled by number of lanes and number of short vectors.

4.4 Matrix - Vector Multiplication

Level 2 BLAS routines typically perform matrix-vector operations with *O (N*^{2}*)* operations per call. Matrix-vector multiply is one of the important kernels in BLAS and there are two basic forms: V^T x M and M x V. It can be formulated as:

$$y_i = \sum_{j=0}^{j=N-1} A_{ij}\, x_j \ \ (i = 0,1,2,3,\dots,N-1)$$

```
MV_product_kernel()
{
 loop (i = 0 to i = N-1)
        $y_i$ = DOT_product_kernel($A_i$,x);
        store result $y_i$ to local memory;
end loop;
}
```

Figure 4.6 Pseudo Code for Matrix-Vector Multiplication

The dot product within the matrix-vector multiply is assigned to one lane, V^T x M performs multi-column access, while M x V performs multi-row accesses to the matrix. As the FPVC can support both accesses with rake type with equal efficiency we chose to implement M x V. Figure 4.6 shows pseudo code for the Matrix-Vector product and as described in the figure DOT product kernel is repeated for every row of matrix. As we scaled up number of vector lanes in FPVC multiple rows of matrix can be computed in parallel.

ANALYSIS: As we have discussed for the vector DOT product, if the maximum vector length (MVL) of the FPVC can accommodate the complete vector in a single vector scalar register, then the performance for each configuration of the FPVC will be similar. Hence, figure 4.7 shows only different lane configurations with constant short vector size of 32.

- For square matrix size of N x N, matrix-vector multiplication performs $2N^2$ floating point operations. It is N times higher than the number of DOT product floating point operations. FPVC underperforms for the square matrix size of 4 and 8 due to prominent impact of main memory access for instruction and data. Highest performance improvement is 1.4x over PowerPC.

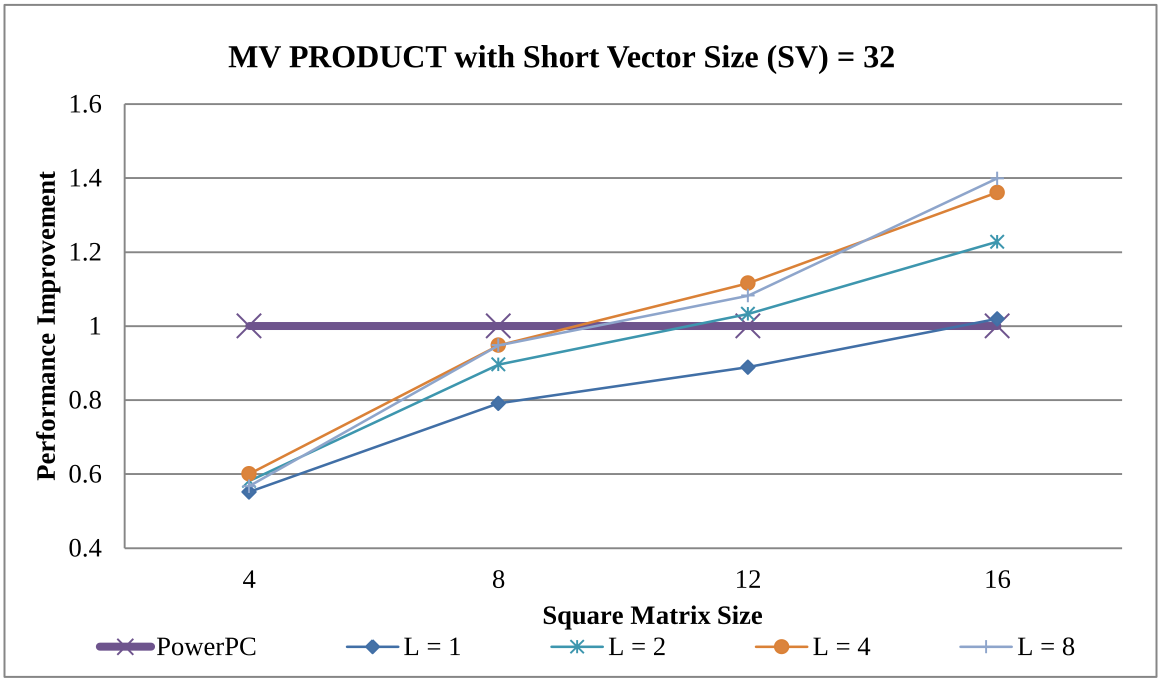


Figure 4.7 Matrix Vector Performance for Lane scaling

4.5 Matrix – Matrix Multiplication

BLAS level 3 routines are primarily matrix–matrix operations that perform $O(N^3)$ arithmetic operations per call. Matrix-matrix multiply is the most important routine within BLAS and there are three basic forms: C = A x B, C = A x B^T, C = A^T x B. This BLAS operation is formulated as:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik}\, B_{kj} \;\; (i, j = 0,1,2,3, \dots, N-1)$$

Matrix – Matrix multiplication can be performed by applying mv_product_kernel to matrix A with each column of matrix B. Figure 4.8 shows pseudo code for C = A x B.

```
MM_product_kernel()
{
 loop (i = 0 to i = N-1)
     $C_i$ = MV_product_kernel(A,$B_i$);
     store result $C_i$ to local memory;
end loop;
}
```

Figure 4.8 Pseudo Code for Matrix-Matrix Multiplication

ANALYSIS: Each element of A and B is used N times and the total number of floating point operations is $2N^3$ which is consistent with the previous two kernel's performance improvements. It underperforms only for the square matrix size of 4 due to significant effect of main memory access for instruction and data. Highest performance improvement is 1.65x over PowerPC.

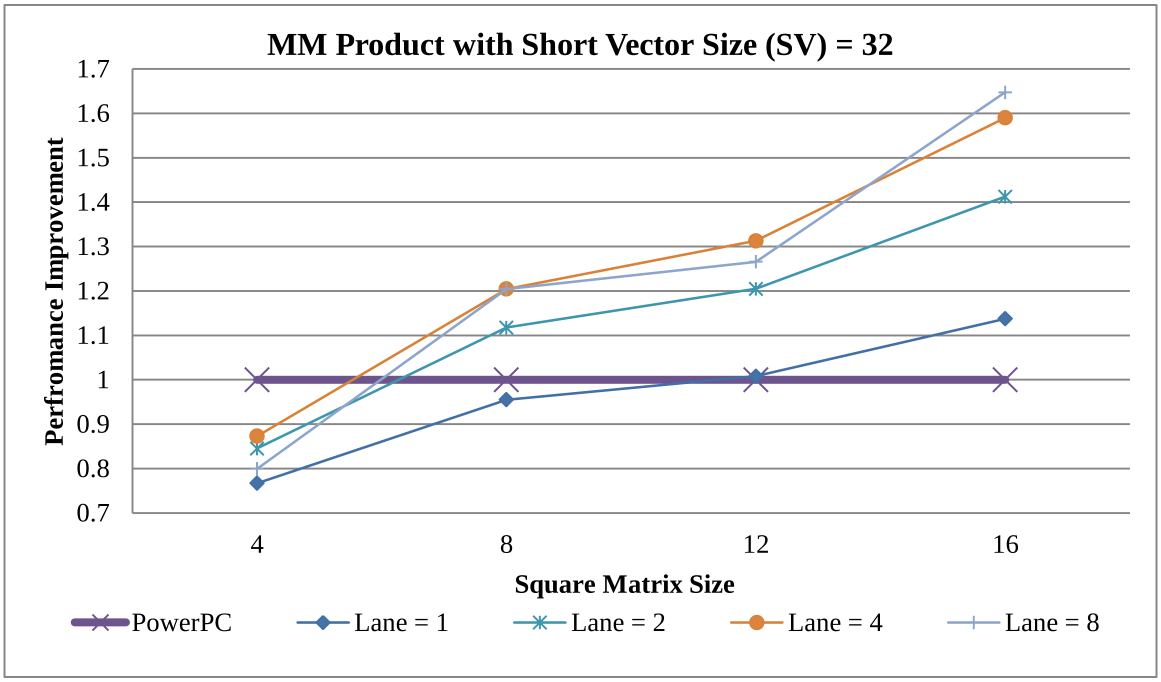


Figure 4.9 Matrix-Matrix Multiplication for Lane Scaling

4.6 QR Decomposition Using Givens Rotation

In linear algebra, a QR decomposition of a matrix is a decomposition of the matrix into an orthogonal and an upper triangular matrix. QR decomposition is often used to solve the linear least squares problem. *QR* decompositions can be computed with a series of Givens rotations. Each rotation zeros an element in the sub-diagonal of the matrix, forming the *R* matrix. The concatenation of all the Givens rotations forms the orthogonal *Q* matrix. A Brief description of the Givens algorithm is given below.

An M x N matrix A is zeroed out one element at a time using 2 x 2 rotation matrix:

$$Q_{ij} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \ where\ i = (2 \ldots M) and\ j = (1 \ldots N)$$

Each rotation matrix $Q_{i,j}$ is orthogonal and will zero out the element A_{ij} in matrix A, starting at the bottom of the first column and working up the columns, then move to the second column and so on. A series of rotation matrices $Q_{i,j}$ are applied to the original matrix A until it becomes an upper triangular matrix R. Here c and s are computed using the following formula:

$$c = \frac{x}{\sqrt{x^2 + y^2}} \ and\ s = \frac{-y}{\sqrt{x^2 + y^2}}$$

```
QR_Decomp_kernel()
{
loop (i = 0 up to i = M-1)
        loop (j = N-1 down to j > i)
                x = A[j-1] [i];
                y = A[j][i];
                compute $Q_{i,j}$;
                A[j-1:j][0:N-1] = MM_product_kernel ($Q_{i,j}$ , A[j-1:j][0:N-1]);
        end loop;
end loop;
}
```

Figure 4.10 Pseudo Code for QR Decomposition

ANALYSIS: Each rotation matrix computation requires 2 multiplications, 1 addition, 1 square root and 2 division operation which total at 6 floating point operations. Also, the MM_product_kernel which multiplies the (2x2) matrix with a (2xN) matrix requires 8N floating point operations. Hence, each rotation of matrix requires 8N+6 floating point operations. So, the total number of floating point operations is (M*(M+1)*(8N+6)/2) which is much higher than matrix multiplication.

Figure 4.11, demonstrates performance improvement for various configuration of the FPVC over the PowerPC. As we discussed in previous three kernels, the more data parallelism available the more performance improvement is observed. Even though, we are using the vectorizable Matrix multiplication kernel, the performance improvement is much higher because of the ample amount of floating point operations. Maximum performance improvement is 4.38x.

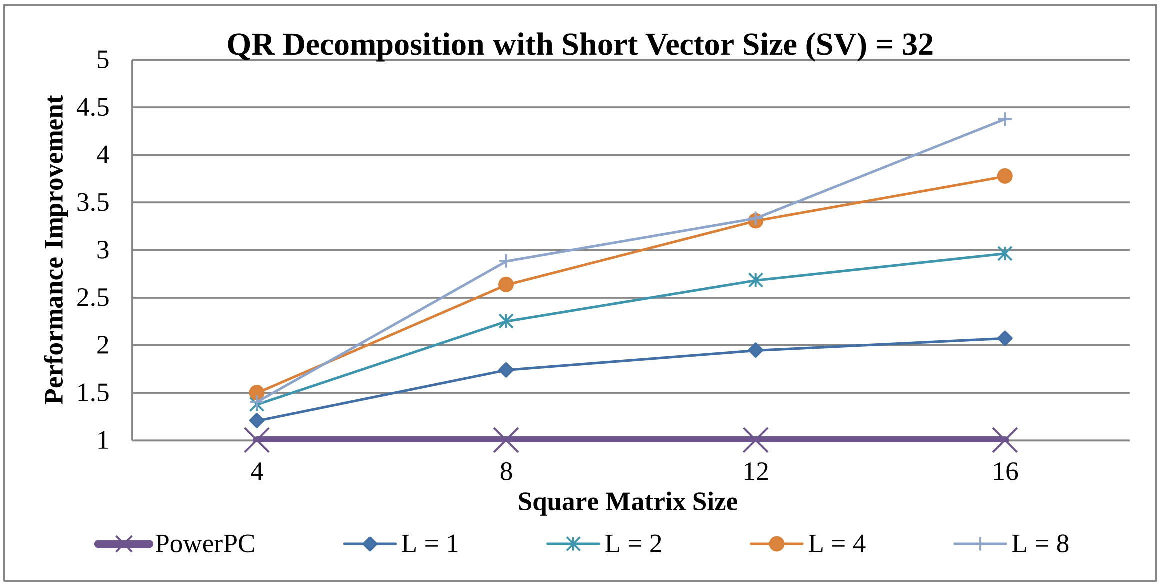


Figure 4.11 QR Decomposition Performance for Lane Scaling

4.7 Cholesky Decomposition

Another LINPACK [41] matrix kernel, Cholesky decomposition, is a decomposition of a symmetric positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose [13]. Cholesky decomposition is mainly used for the numerical solution of systems of linear equations.

If matrix A has real entries and is symmetric and positive definite, then matrix A can be decomposed as,

$$\boldsymbol{A = L * L^T}$$

where L is a lower triangular matrix with strictly positive diagonal entries and L^T denotes the conjugate transpose of L. Each element of matrix L can be defined as below:

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} \text{ and } L_{i,j} = \frac{1}{L_{j,j}}\left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}\, L_{j,k}\right) for\ i > j$$

```
Cholesky_Decomp_kernel()
{
 loop (i = 0 up to i = N-1)
        pivot value = sqrt ($A_{i,i}$);
        divide i^th column vector from i to N by pivot value;
        loop (j = i+1 upto N)
                accumulate row vector from 0 to i;
                subtract accumulated value from $A_{j,i+1}$;
        end loop;
 end loop;
}
```

Figure 4.12 Pseudo code for Cholesky Decomposition

The computation is usually arranged in either of the following orders.

- The one starts from the upper left corner of the matrix *L* and proceeds to calculate the matrix row by row.
- Another which we use, starts from the upper left corner of the matrix *L* and proceeds to calculate the matrix column by column.

As shown in the pseudo code, first we find the square root of the pivot value which is element $A_{i,i}$. Then divide column vector which contains elements $A_{i,i}$ to $A_{N,i}$. Next we accumulate the row vector $A_{i,0:i}$ and subtract the accumulated value from each element of the row vector $A_{i,i+1:N}$ which contains elements from i to N by pivot value. Every iteration of Cholesky Decomposition will compute result column vector and updates elements, which are right side to the pivotal column.

ANALYSIS: We can only compute the $A_{i,j}$ entry if we know the entries to the left and above. Hence each column or row vector cannot be assigned to separate vector lanes and compute independently. This can be observed in the figure 4.11 as scaling of vector lane did not improve performance significantly. Hence, a major criterion for the performance improvement is not number of vector lane but the maximum vector length (MVL). Minimum performance improvement for short vector size of 32 is 3x. The maximum performance improvement is 4.3x.

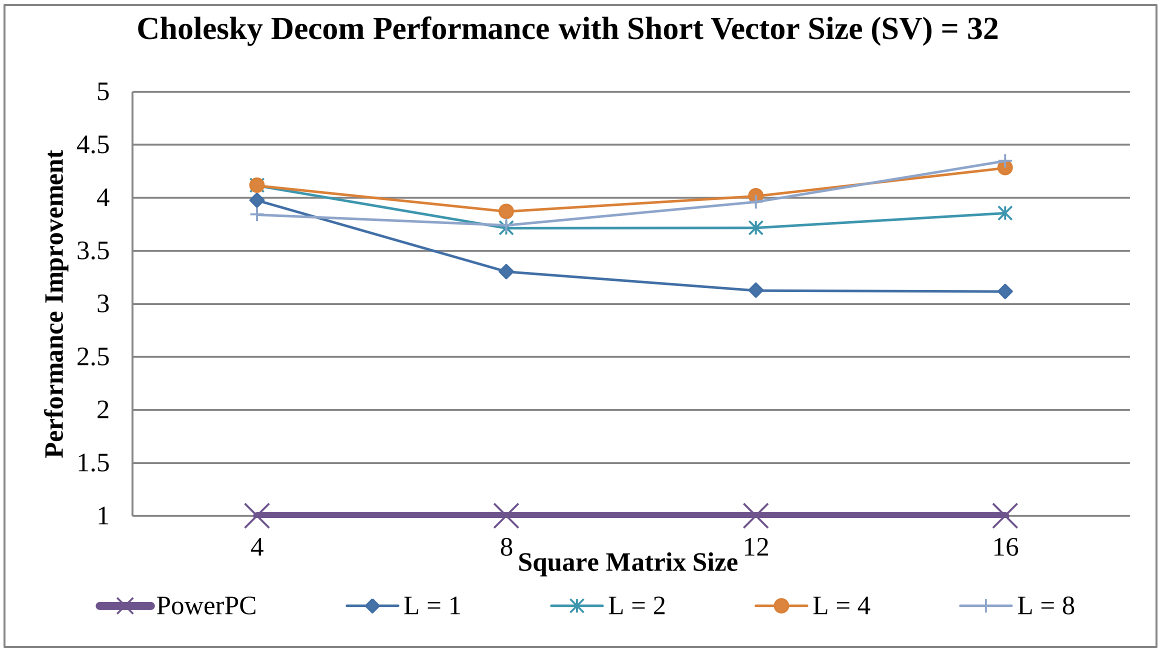


Figure 4.13 Cholesky Decomposition Performance for Lane Scaling

4.8 Area requirement

Table 4.1, lists area usage after Place and Route of embedded processor design.

Floating Point Configuration	Xilinx FPU	SV=8 L=1	SV=16 L=1	SV=32 L=1	SV=32 L=2	SV=32 L=4	SV=32 L=8
# of LUT slices	3,055	5,414	5,453	5,493	9,013	15,628	26,196
Block RAMs	16	51	51	51	54	60	72
Multipliers	3	13	13	13	26	52	104

Table 4.1 Area Usage For Different FPVC Configuration

As shown in table 4.1, by doubling the short vector size the LUT requirement increases by merely 1% whereas by doubling the number of lane the LUT requirement increases on an average by 70%. For the comparable design of Xilinx's FPU and FPVC the LUT usage increase by nearly 100%.

4.9 Conclusion

In this chapter we have described vectorized linear algebra kernels and evaluated performance of each kernel with respect to PowerPC. For various small vector and/or matrix multiplication, main memory access latency effects are prominent on the result and they are underperformed compared to PowerPC. Also, for the matrix and vector kernels, where each vector computation can be mapped on vector lane, vector lane scaling is an ideal option but as seen in Cholesky decomposition kernel, where each matrix element computation depends on the previous computations lane scaling will be waste of resources. By designing parameterisable FPVC, we are able to tune the design resources as per the nature of the kernel and hence we can better optimize area versus performance.

5 CONCLUSIONS AND FUTURE WORK

In this thesis, we have the described design and implementation of a Floating Point Vector Co-processor (FPVC) on an FPGA. This is one of the first works which implements floating point arithmetic with Hybrid Vector/SIMD co-processor architecture on the FPGA. FPGA provides a unique feature, the reconfigurability of the design, with which we can tune our FPVC according to the amount of parallelism available within floating point arithmetic applications.

All the previous work in the design of vector processor have implemented a vector processor as an extension to the scalar processor and these vector processors still fully depends on the scalar processor for instruction fetch and partially depends for instruction decoding. Register file and computational resources are duplicated for scalar and vector processors and hence require more area. In this work, we have implemented a unified approach where all scalar and vector instructions truly share all resources in the design. Hence, our approach is more area efficient compared to others.

We have initiated a floating point library design for linear algebra kernels, which can adapt various configurations of the FPVC and tune software library. In this manner, not only FPVC hardware design but software library is also tuned for optimal performance. We have analyzed vectorization of these kernels with respect to PowerPC based kernel execution and deduced that by scaling short vector registers we can improve overall performance of each kernel without significant increase in area usage. Also, the data level parallel tasks which can be independently mapped on each vector lane scales performance with number of vector lane scaling. Finally, the more compute intensive applications show the most improved perfromance.

FUTURE WORKS

There are several avenues of research which lead on from this work:

1) **Variable Precisions Floating Point Data Support:** All the floating point computational units are derived from VFLOAT [16] library which supports variable precision and hence variable data width. By limiting floating point precision and so as the data width, we can improve area efficiency of the FPVC.

2) **Architectural Improvements:** As BRAM has only two ports for memory accesses, currently our design does not support vector chaining but in future multiple BRAM ports can improve the performance of the FPVC. Also, the inclusion of a unified or separate instruction and data cache can improve performance.

3) **Application Vectorization:** In this research we have started designing a library of parallel vectorized basic linear algebra kernels. Many more Digital Signal Processing, multimedia kernels which exhibits data level parallelism can be added to our library design.

4) **Advanced Vector Compilation:** All the kernels are hand written using our vector scalar instruction set and are not optimized. Even the vectorized kernel exhibits speed up over the PowerPC for most of the data sets, they can benefit from further advances in vector compiler technology. Several new vector compilation challenges are introduced by the architectural ideas in this work, including reconfigurability of lanes, Hybrid vector/SIMD architecture and data management in a unified vector-scalar architecture.

5) Multi-core Design Exploration: As FPVC can work autonomous from main processor, inclusion of features related to multi-processor support can improve a performance of broader application base.

REFERENCES

[1] “Nios II Processor Reference Handbook.” http://www.altera.com/literature/hb/nios2/ n2cpu nii5v1.pdf, 2009.

[2]“MicroBlaze Processor Reference Guide.” http://www.xilinx.com/support/ documentation/sw manuals/mb reference guide.pdf, 2008.

[3] “Embedded Processor Block in Virtex-5 FPGAs.” http://www.xilinx.com/support/ documentation/user guides/ ug200.pdf, 2009.

[4] “Virtex-5 APU Floating-Point Unit v1.01a.” http://www.xilinx.com/support/ documentation/ip documentation/apu fpu virtex5.pdf, April 2009. Data Sheet DS693.

[5] P. Yiannacouras, J. Gregory Steffan, and Jonathan Rose, VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors, International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), October 2008, Atlanta, GA.

[6] J. Yu, G. Lemieux, and C. Eagleston, "Vector Processing as a Soft-core CPU Accelerator," ACM International Symposium on FPGA, 2008.

[7] “ASIC or FPGA: Why Not Plan For Portability?”, By Colin Baldwin and Ehab Mohsen, http://chipdesignmag.com/display.php?articleId=3182.

[8] V. Betz, J. Rose, and A. Marquardt, Eds., Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, 1999.

[9] IEEE Standards Board and ANSI, IEEE Standard for Binary Floating- Point Arithmetic. IEEE Press, 1985. IEEE Std 754-1985.

[10] Steven W. Smith, “The Scientist and Engineer’s Guide to Digital Signal Processing”, Second Edition, Chapter 28 Digital Signal Processor.

[11] G. Lienhart, A. Kugel, and R. Manner. Using floating-point arithmetic on FPGAs to accelerate scientific N-Body simulations. In 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’02), pages 182–191, April 2002.

[12] B. Lee and N. Burgess. Parameterisable floating-point operations on FPGA. In 36th Asilomar Conference on Signals, Systems and Computers, IEEE Signal Processing Society, November 2002.

[13] G. Govindu, R. Scrofano, and V. K. Prasanna. A library of parameterisable floatingpoint cores for FPGAs and their application to scientific computing. The 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2005), June 2005.

[14] R. Matousek, M. Tich, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman. Logarithmic Number System and Floating-Point Arithmetics on FPGA. In Lecture Notes in Computer Science, volume 2438, pages 175–188. Springer, 2002.

[15] A VHDL library of parametrisable floating-point and LNS operators for FPGA. http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/

[16]"Northeastern University Variable Precision Floating-Point Modules." http://www.ece.neu.edu/groups/rcl/projects/floatingpoint/.

[17] X. Wang, S. Braganza, and M. Leeser, Advanced Components in the Variable Precision Floating-Point Library, IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM2006), pp. 249-258. April 2006.

[18] B. Fagin and Cyril Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic", IEEE Transactions on VLSI Systems, Volume 2, September 1994.

[19] Zhanpeng Jin, Richard Neil Pittman, and Alessandro Forin, Reconfigurable Custom Floating-Point Instructions, Microsoft Research, no. MSR-TR-2009-157, August 2009

[20] P. Diniz and G. Govindu, "Design of a Field Programmable Dual – Precision Floating Point Arithmetic Unit", International Conference on Field Programmable Logic and Applications (FPL), 2006.

[21] M. Schulte and E. Swartzlander, Jr., "Hardware Design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor", Symposium on Computer Arithmetic, 12th proceeding, 1995.

[22] R.E. Moore, Interval Analysis. Prentice-Hall, Englewood Cliffs, N.J.. 1966.

[23] C. Brunelli, F. Garzia, J. Nurmi, et al., "A FPGA Implementation of an Open – Source Floating Point Computation System", Internation Symposium on System-on-Chip, Page 29-32, 2005.

[24] T. Rodolfo, N. L. V. Calazans, et al., "Floating Point Hardware for Embedded Processors in FPGAs : Design Space Exploration for Performance and Area", International conference on Reconfigurable Computing and FPGAs, 2009.

[25] D. Burke, J. Wawrzynek, K. Asanovic, et al., "RAMP Blue: Implementation of a Manycore 1008 Processor FPGA System," in Reconfigurable Systems Summer Institute (RSSI), July 2008.

[26] V. E. Petcu, A. Amaricai, and M. Vladutiu, “A dual-threaded architecture for interval arithmetic coprocessor with shared floating point units,” in 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, pp. 1–4, 2008.

[27] N. Hockert and K. Compton, “Ffpu: Fractured floating point unit for fpga soft processors,” in Field ProgrammableTechnology (FPT), pp. 621–624, 2009.

[28] J. Kadlec, R. Bartosinski, and M. Danek, “Accelerating microblaze floating point operations,” in Field Programmable Logic and Applications (FPL), pp. 621–624, 2007.

[29] H. Yang and S. Ziavras, “FPGA based Vector Processor for Algebraic Equation Solvers”, IEEE International SOC conference, 2005, Page 115-116.

[30] S. Chen, R. Venkatesan, P. Gillard, “Implementation of Vector Floating-Point Processing Unit on FPGAs For High Performance Computing”, Canadian conference on Electrical and Computer Engineering, 2008. Page 881- 886.

[31] C. Kozyrakis, “Scalable Vector Media Processors for Embedded Systems,” Ph.D. dissertation, University of California-Berkeley, 2002.

[32] R. G. Hinta and D. P. ‘Pate. Control data STAR-100 processor design. In Proc. Compcon 72, pages 1-4, New York, 1972. IEEE Computer Society Conf. 1972, IEEE.

[33] R. M. Russell. The CRAY-1 computer system. Communications of the ACM, 21(1):63-72, January 1978.

[34] L. Huang and Z. Wang, “SV: Enhancing SIMD Architectures via Combined SIMD-Vector Approach”, In ICA3PP 2010, Part I, pp. 226-235.

[35] C. Kozyrakis and D. Patterson, “Overcoming the Limitations of Conventional Vector Processors”, In Proceedings of the 30th International Symposium on Computer Architecture, San Diego, California, June 2003, pp. 399–409.

[36] R. Espasa and M. Valero, “Decoupled Vector Architecture”, In the Proceedings of the 2nd Intl. Symp. On High-Performance Computer Architecture, Pages 281–90, San Jose, CA, Feb. 1996.

[37] R. Espasa, M. Valero, and J. Smith, “Out-of-order Vector Architectures”, In the Proceedings of the 30th Intl. Symp. On Microarchitecture, Pages 160–70, Research Triangle Park, NC, Dec. 1997.

[38] Suzanne Rivoire, Rebecca Schultz, Tomofumi Okuda, Christos Kozyrakis, “Vector Lane Threading”, Proceedings of the 2006 International Conference on Parallel Processing (ICPP’06).

[39] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, and N. Morgan, “The T0 Vector Microprocessor,” Hot Chips, vol. 7, pp. 187–196, 1995.

[40] Silviu Ciricescu , Ray Essick , Brian Lucas , Phil May , Kent Moat , Jim Norris , Michael Schuette , Ali Saidi, “The Reconfigurable Streaming Vector Processor”, 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. Page 141 – 150.

[41] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users’ Guide*. Society for Industrial and Applied Math., 1979.

[42] C. Lawson, R. Hanson, D. Kincaid and F. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. ACM Transaction on Mathematical Software, 5(3): 308-323, 1979.

[43] ML510 Reference Design User Guide, “http://www.xilinx.com/support/ documentation/boards_and_kits/ug355.pdf”, 2009.

[44] Fabric Co-processor Bus for PPC440 (FCB_V20) (v1.00a), Product Specification, 2008.

[45] L. Zhuo, V.K. Prasanna, “High Performance Linear Algebra Operations on Reconfigurable Systems”, in Proceedings of Supercomputing, November 2005, Washington, USA

[46] X. Wang and M. Leeser, “A Truly Two Dimensional Systolic Array FPGA implementation of QR Decomposition,” ACM Transactions on Embedded Computer Systems, Vol. 9 No. 1, October 2009.

[47] D. Yang, G.D. Peterson, H. Li, J. Sun, “An FPGA Implementation for Solving Least Square Problem” , IEEE Symposium of FCCM,2009, Page 303-306.

APPENDIX A

A Floating Point Vector Co-processor (FPVC) is a Hybrid vector/SIMD (Single Instruction Multiple Data) co-processor. Maximum vector length of the co-processor is decided by the number of vector lanes within each short vector and the number of short vectors within each general purpose registers. The vector scalar instruction set architecture, which borrows heavily from the VIRAM instruction set architecture, is described in chapter 3. The details of the instruction set are described in this appendix.

A.1 Data Types

32-bit unsigned integer and single precision floating point data are supported by the processor. One exception is integer multiply operation which is implemented for 16-bit width.

A.2 Addressing modes

Instruction set supports both scalar and vector memory accesses. There are two types of addressing modes supported for scalar memory accesses:

1. Immediate accesses
2. Indexed accesses

Vector memory accesses are supported with below addressing modes:

1. Unit – stride accesses
2. Non unit-stride accesses
3. Indexed offset accesses with no constant immediate value
4. Indexed offset accesses with constant immediate value

A.3 Instruction Classes

All the instructions can be classified in below categories:

1. Memory access instructions
2. Integer arithmetic instructions
3. Program flow control instructions
4. Floating point arithmetic instructions
5. Special instructions

A.4 Instruction References

Below table describes keywords used for instructions in next section.

Name	**Description**
op	*Instruction operation*
Rd	*destination register number*
r1	*source operand1's register number*
r2	*source operand2's register number*
reg-type	*Type of each register,* *sss-all are scalar type* *vss-only desitnation register is vector type* *vvs-destination and operand1 register are vector type, operand2 is scalar type* *vsv-destination and operand2 register are vector type, operand1 is scalar type* *vvv-all are vector type*
exop/imd(11)/imd(16)	*11-bit extended opcode or 16-bit/ 11-bit immediate field*

Table A.1 Instruction References

A.5 Instruction Format

Instruction with 2- source operands

op[5:0]	*rd[4:0]*	*r1[4:0]*	*r2[4:0]*	*exop[10:0]*

Instruction with 1 source operand and 11-bit immediate value

op[5:0]	*rd[4:0]*	*r1[4:0]*	*r2[4:0]*	*imd[10:0]*

Instruction with 1 source operand and 16-bit immediate value

op[5:0]	*rd[4:0]*	*r1[4:0]*	*imd[15:0]*

Conditional jump instruction format

op[5:0]	-	*lt*	*gt*	*eq*	-	-	*imd[15:0]*

Immediate jump instruction format

op[5:0]	-	-	*imd[15:0]*

A.6 Instructions

Operation	Mnemonics	Syntax	Summary
Memory Access Instructions			
Load	ld ld.imd	ld.[reg-type] rd, r1, r2, imd(11) ld.[reg-type] rd, r1, imd(16)	Loads/Stores data to/from the destination register from/to memory location. Base address is provided by r1, r2 provides index and imd(11)/imd(16) provides constant stride for the vector access and immediate value for scalar access
Store	st st.imd	st.[reg-type] rd, r1, r2, imd(11) st.[reg-type] rd, r1, imd(16)	
Integer Arithmetic Instructions			
Addition	add add.imd	add.[reg-type] rd, r1, r2, exop add.[reg-type] rd, r1, imd(16)	adds source operand2 or immediate to source operand1 and stores result at destination
Subtraction	sub sub.imd	sub.[reg-type] rd, r1, r2, exop sub.[reg-type] rd, r1, imd(16)	subtracts source operand2 or immediate from source operand1 and stores result at destination
Multiply	mul mul.imd	mul.[reg-type] rd, r1, r2, exop mul.[reg-type] rd, r1, imd(16)	multiplies source operand1 with source operand2 or immediate and stores result at destination
Shift right logical	srl srl.imd	srl.[reg-type] rd, r1, r2, exop srl.[reg-type] rd, r1, imd(16)	right shifts source operand1 by the amount of source operand2 or immediate and stores result at destination
Shift left logical	sll sll.imd	sll.[reg-type] rd, r1, r2, exop sll.[reg-type] rd, r1, imd(16)	left shifts source operand1 by the amount of source operand2 or immediate and stores result at destination
Compare	cmp cmp.imd	cmp.[reg-type] rd, r1, r2, exop cmp.[reg-type] rd, r1, imd(16)	compares source operand1 to source operand2 or immediate and comparison is stored in status register also for vector mask stores targeted comparison

Program Control Flow Instructions			
Jump	bri brc	bri imd(16) brc cond,imd(16)	Jump from current PC to offset provided in immediate part, conditional jump evaluates status register with condition and jumps to targeted PC
Floating Point Arithmetic Instructions			
Addition	fadd	fadd.[reg-type] rd, r1, r2, exop	adds source operand2 to source operand1 and stores result at destination
Subtraction	fsub	fsub.[reg-type] rd, r1, r2, exop	subtracts source operand2 from source operand1 and stores result at destination
Multiply	fmul	fmul.[reg-type] rd, r1, r2, exop	multiplies source operand1 with source operand2 and stores result at destination
Division	fdiv	fdiv.[reg-type] rd, r1, r2, exop	divides source operand1 by source operand2 and stores result at destination
Square root	fsqrt	fsqrt.[reg-type] rd, r1, r2, exop	stores square root of operand 2 at destination
Compare	fcmp fcmp.imd	fcmp.[reg-type] rd, r1, r2, exop	compares source operand1 with source operand2 and comparison is stored in status register also for vector mask stores targeted comparison
Special Instructions			
SPR move	mtr mfr	mtr rd, r1, imd(16) mfr rd, r1, imd(16)	moves special purpose registers value to/from general purpose register
NOP	nop	nop	No operation
HALT	halt	halt	Halts execution

Table A.2 Instructions

A.7 Instruction opcode encoding

Table A.3 lists the opcode field encodings for vector scalar instructions whereas table A.4 lists the encoding of extended opcodes.

	opcode[5:2]							
opcode [1:0]	0000	0001	0010	0011	0100	0101	0110	0111
00	nop		mtr		add.imd	srl.imd	ld.sss	
01	halt		mfr		sub.imd	cmp.imd	st.sss	
10			brc		mul.imd	ld.imd	int.sss	
11			bri		sll.imd	st.imd	flt.sss	

Table A.3 (a) opcode encoding

	opcode[5:2]							
opcode [1:0]	1000	1001	1010	1011	1100	1101	1110	1111
00	ld.vss		ld.vsv		ld.vvs		ld.vvv	
01	st.vss		st.vsv		st.vvs		st.vvv	
10	int.vss		int.vsv		int.vvs		int.vvv	
11	flt.vss		flt.vsv		flt.vvs		flt.vvv	

Table A.3 (b) opcode encoding

exop [10:0]	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9
operation	cmp	add	sub	mul	div	sqrt	sll	srl	cmprs	expand

Table A.4 extended opcode encoding