Microprocessors and Microsystems (Elsevier journal), accepted for publication.

Versatile Design of Shared Vector Coprocessors for Multicores

Spiridon F. Beldianu, Christopher Dahlberg, Timothy Steele and Sotirios G. Ziavras
Electrical and Computer Engineering Department
New Jersey Institute of Technology
Newark, NJ 07102
ziavras@njit.edu

Abstract— *For most of the applications that make use of a vector coprocessor, its resources are not highly utilized due to the lack of sustained data parallelism, which often occurs due to insufficient vector parallelism or vector-length variations in dynamic environments. The motivation of our work stems from (a) the omnipresence of vector operations in high-performance scientific and emerging embedded applications; (b) the mandate for multicore designs to make efficient use of on-chip resources for low power and high performance; (c) the need to often handle a variety of vector sizes; and (d) vector kernels in application suites may have diverse computation needs. Our objective is to provide a versatile design framework that can facilitate vector coprocessor sharing among multiple cores in a manner that maximizes resource utilization while also yielding very high performance at reduced area and energy costs. We have previously proposed three basic shared vector coprocessor architectures based on coarse-grain temporal, fine-grain temporal, and vector lane sharing that were implemented in SystemVerilog [15]. Our new paper presents substantially improved versions of these architectures that are implemented in synthesized RTL for higher accuracy. We herein evaluate these vector coprocessor sharing policies for a dual-core system using the floating-point performance, resource utilization and power consumption metrics. Benchmarking for FIR filtering, FFT, matrix multiplication, LU decomposition and sparse matrix vector multiplication shows that these coprocessor sharing policies yield high utilization, high performance and low energy per operation. Fine-grain temporal sharing most often provides the best performance among the three policies; it is followed by vector lane and then coarse-grain temporal sharing. It is also shown that, per core exclusive access to the vector resources does not maximize their utilization. This benchmarking involves various scenarios for each application, where the scenarios differ in terms of the vector length and the parallelism-oriented coding technique.*

Keywords- *Vector coprocessor, coprocessor sharing, multicore, FPGA prototyping.*

1. Introduction

SIMD architectures are very efficient for multimedia data processing and scientific applications because they can process simultaneously multiple data elements based on one vector instruction. VIRAM [3], SODA [7] and AnySP [4] are single-chip vector microprocessors; i.e., their instruction sets support vector operations by comprising a comprehensive vector architecture. Vector microprocessors have been shown to be more effective in embedded media applications than superscalar and VLIW processors [1]. Also, a 2-dimensional (matrix) SIMD extension was developed in [13]. Due to recent advances in programmable devices that have substantially increased their logic cell densities, some FPGA-based soft vector processors have been proposed as well [6-10].

Nevertheless, many high-performance and embedded applications dealing with streams of data cannot fully utilize dedicated vector processors for various reasons. Firstly, individual programs often display limited percentage of vector code due to substantial flow control or operating system tasks. The utilization of the *Vector coProcessor* (VP) is then proportional to the vectorized part of the code and the rest of the time the VP will be idle [11]. Secondly, even with substantial vector code, some applications deal with small-sized vectors. In fact, the vector length often varies across algorithms or within a single algorithm, as in multimedia applications [4]. Thirdly, several applications have many data dependencies in sequences of instructions, which are exacerbated further when loop unrolling and/or other advanced scheduling techniques are not applied [12]. Such limitations strangle efforts to sustain a high utilization for the vector unit, especially with the inclusion of advanced pipelining for floating-point operations.

Moreover, static power due to leakage current will become an even larger source of power consumption in future technologies. The shrinking of transistors increases the static power contribution to the total energy [14]. Therefore, with a lower utilization of resources, the static energy becomes a larger component of the power budget. Thus, increasing the utilization of resources will reduce the average energy consumption per operation.

We could increase the overall utilization and throughput of a vector unit embedded into a multicore chip by providing a mechanism for the simultaneous sharing of the vector processor by instructions issued by multiple scalar processors. The terms scalar processor and core processor will be used interchangeably from now on. This approach could support multithreading inside the vector unit, where the threads come from either a single or multiple applications running on the multicore chip. Unlike conventional VP architectures for single cores which are designed with a fixed SIMD width (i.e., vector register size) aiming to service any vector case presented by applications, we suggest that a shared VP for multicores should support

multiple-SIMD execution relying on thread-level parallelism (TLP). Such a vehicle will allow us to maximize the utilization and throughput of the VP for two reasons: (a) different cores often deal with different vector lengths, thus not being able to individually utilize the VP resources fully; and (b) different vector kernels in the same or different applications often have diverse VP-based computation needs [4]. To alleviate the aforementioned drawbacks while also facilitating the release of on-chip real estate for other important design choices, we proposed in [15] an adaptable VP sharing scheme for multicores that integrates three basic VP sharing architectures: coarse-*grain temporal (CTS) sharing, fine-grain temporal sharing (FTS), and vector lane sharing (VLS).* Our system was implemented in the SystemVerilog high-level language and only performance benchmark results were recorded. Here we present an improved VP sharing integration that, besides substantial architectural improvements and new vector instructions, is implemented on an FPGA in synthesizable VHDL for highly accurate benchmarking results.

CTS sharing consists of temporally multiplexing sequences of vector instructions or threads containing them. Each scalar/core processor takes exclusive control of the VP by executing lock/unlock instructions, and runs a thread to completion or until a vector instruction has to stall for many cycles (e.g., for operand fetching or resource conflicts). Such a stall forces thread switching for the VP.

FTS sharing consists of temporally multiplexing individual vector instructions coming from different scalar processors. Hence, each scalar processor gets hold of the VP according to a chosen arbitration scheme, the simplest one being round robin. The benefit of this approach is that the VP utilization will be increased since data hazards do not exist between instructions issued by different processors, and the idle times due to data transfers are eliminated or reduced. However, FTS requires per core vector register renaming. This approach can be extended to implement simultaneous multithreading for vector operations, where vector instructions coming from different cores can simultaneously execute in the same VP as long as they utilize different vector resources (e.g., adder and multiplier) [10].

VLS lane sharing assumes a divisible VP consisting of independent vector lanes with their own execution units. A vector lane is an independent vector sub-unit containing its own bus interfaces, processing units and vector registers; during its operation it does not compete for resources with any other lane, except for external accesses to the same memory modules. VLS facilitates the simultaneous allocation of distinct vector lanes, or collections of them, to distinct scalar processors for seamless processing. Based on the chosen set of vector-lane allocation and scheduling policies, a lane-external hardware scheduler decides how to split the vector lanes or combine them based on the requirements of the applications running on the cores. Therefore, if multiple cores are simultaneously assigned the VP space, each scalar processor can use exclusive coprocessor resources forming a small-sized VP (as compared to the full-sized VP that comprises all of the lanes, say M). Assuming that each lane contains n elements per vector register, a small-sized VP with m lanes, where $m<M$, can operate on vector registers having $m \times n$ elements. VLS proves useful when the degree of vectorization required by an application running on a core is moderate, thus not requiring the full coprocessor space, or when there are multiple scalar processors requiring vector processing at the same time. Also, VLS could be extended to cases where each small-sized VP simultaneously supports two or more threads coming from the same or different cores. However, this approach will be investigated in future work.

The main difference of our work from [3], [4] and [9] consists of introducing (a) a hybrid architecture for vector processor design that can facilitate the coarse- and fine-grain mixing of threads coming from multiple cores, as well as (b) reconfigurable vector lanes that can form single or multiple VPs for assignment to distinct cores in a manner that eliminates internal resource conflicts. We should emphasize that no other work related on Vector Coprocessor sharing has been proposed so far. Also, our main objective as compared with all previous aforementioned work, where just one thread can use the entirety of the VP resources, is to provide a hybrid architecture framework for sharing a Vector coProcessor among multiple scalar cores. This architecture is especially suitable for shared-bus multicores, the current focus of commercial technology. This paper uses the terms vector processor and SIMD architecture interchangeably from now on. Although our emphasis is on vector-processor design to support its simultaneous sharing among various processor cores, the fact that different cores can initiate different vector instruction sequences makes our design capable of more general SIMD processing.

The major differences of this paper from our earlier work [15, 20] are: (a) Actual implementations on an FPGA using synthesizable VHDL (instead of higher-level SystemVerilog); (b) a larger number of benchmarked applications involving also many more scenarios; (c) the inclusion of fused multiply-add (MADD) and divide (VDIV) instructions in the vector lanes; (d) the production of power and energy consumption results followed by a relevant analysis; (e) scalability analysis for various configurations of the vector coprocessor involving 2, 4, 8, 16 and 32 lanes; (f) performance results for random scenarios involving two threads that contain vector kernels interleaved with idle times; and (g) synthesis frequency scalability analysis.

The rest of the paper is organized as follows. Section 2 presents the three architectures for the basic vector-sharing design choices. Section 3 contains benchmarking results involving the performance and power metrics, as well as comparative and scalability analyses. Finally, conclusions are drawn in Section 4.

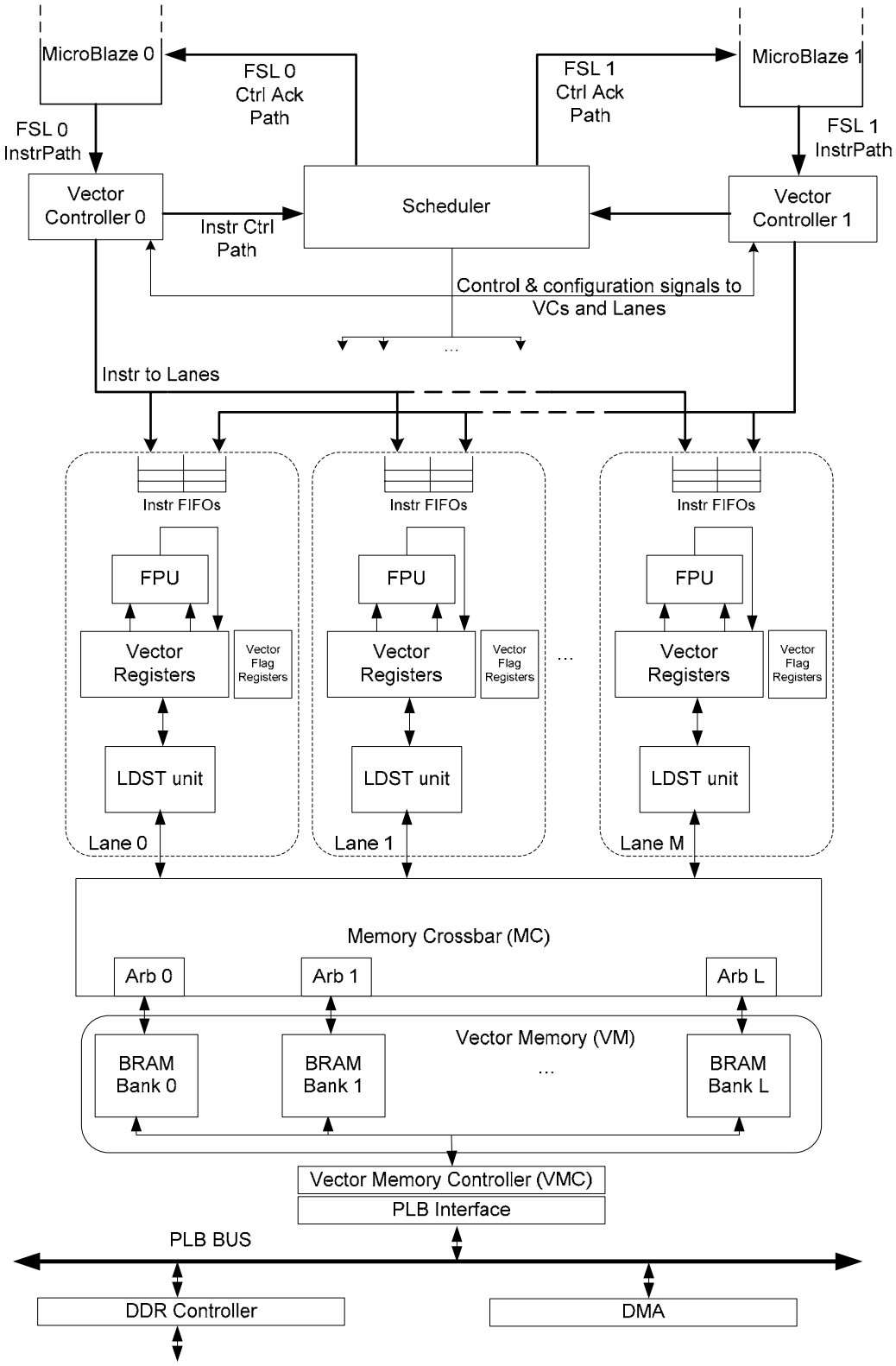

Figure 1. VP sharing architecture (PLB: Xilinx Processor Local Bus, used mostly for data transfers via DMA; Xilinx FSL: Fast Simplex Link FSL serves as the instruction path between a MicroBlaze and its associated Vector Controller, through the Scheduler; BRAM: Xilinx Block RAM; each MUX in the figure is part of the respective lane).

2. VP-Sharing Architectures

2.1. VP Architecture

2.1.1. General Overview

In order to validate the vector-sharing contexts under our FTS, CTS and VLS design schemes, we have built an FPGA-based system that consists of two scalar processors, an *M*-way data-path partitioned VP with an *M*-way vector memory load/store unit for parallel data memory accesses, a VP-memory interconnecting crossbar, and an *L*-bank low-order interleaved on-chip vector memory. Our VP uses the VIRAM lane-based concept [1-3] that was proposed for a single core. With our design, however, the vector lane space can be partitioned among multiple cores as needed. This adaptable structure can even be used to assign variable numbers of vector lanes to the cores throughout execution based on application needs, as per the VLS design choice. Each vector lane contains a subset of the elements from a larger vector register, one floating-point unit (FPU) and a memory load/store (L/S) unit (LDST). Figure 1 presents the complete system prototype. The Vector Processor (VP), Memory Crossbar (MC), Vector Memory (VM) and Vector Memory Controller (VMC) are custom IPs that were designed in VHDL, and the rest of the system was generated using the Xilinx EDK tool, version 12.3.

A Vector Controller (VC), which is attached to a scalar processor, has a latency of two clock cycles; one clock cycle is needed for decoding and hazard detection, and one more clock cycle for register renaming and scheduling. After decoding and hazard detection, the Vector Controller broadcasts the vector instruction to its assigned lanes by pushing it with the appropriate vector element ranges into small instruction FIFOs located in the respective lanes. Inside a lane, a state machine takes the instruction from the FIFO, decodes it, fetches the appropriate operands, and issues them to the appropriate execution unit. Each lane assigned to a VC informs it upon instruction completion and the entire SIMD instruction is considered completed by the VC when all the lanes have completed this step. Each SIMD instruction is labeled by the VC with a unique tag and dedicated signaling from the lane to its assigned VC informs the latter about the completion of the instruction.

The elements corresponding to each vector register are distributed across multiple lanes in the low-order interleaved fashion and the number of elements from a vector register corresponding to a lane is configurable. Each lane contains four configuration registers which are updated at runtime by the Scheduler. The first register contains the VC ID to which the lane is assigned while the second register contains the number of lanes assigned to the particular VC to which this lane is assigned. This register is updated when switching between the CTS/FTS and VLS operating contexts. The third register contains the fixed lane ID (or lane index). The fourth register contains the number of elements from a vector register which are located in the same lane. Since the Vector Register File (VRF) within each lane has fixed size, increasing the number of elements in a vector register will automatically decrease the number of available registers.

Besides various miscellaneous operations having one clock cycle latency, the execution unit in each lane also includes a multiply and add/subtract single-precision floating-point units which are highly pipelined. The LDST unit from each lane can operate with or without a vector stride, and can also carry out indexed memory accesses using the crossbar going to the memory. The crossbar allows for concurrent accesses from LDST units to distinct memory banks and also provides round-robin arbitration when many LDST units are accessing the same memory bank.

2.1.2. FPGA Implementation

The system was implemented on Virtex-5 FPGA using the Xilinx ISE tools. The processors are identical based on MicroBlaze, a 32-bit embedded RISC soft core provided by Xilinx. It employs the Harvard architecture and uses the FSL interface to connect with up to eight coprocessors [5]. Instructions issued to the Vector Processor in our design use a 32-bit lane of the FSL bus. Since the Xilinx EDK tool kit limits the operating frequency of the MicroBlaze processor to 125 MHz, we optimized the entire design for this frequency. The multi-ported VRF with 512 32-bit locations efficiently implemented with Xilinx FPGA 36Kbit BRAMs (Block RAMs). Each of the LDST and ALU units requires two reads and one write per clock cycle. Two read ports are needed by store instructions involving an index vector register, where one port is used to access the indices and the other one to get the elements to be stored. Therefore, the memory has two write and four read ports (2W/4R), and is implemented by doing replication (2×) and multi-pumping using a double frequency [16]. The multiply and add/subtract pipelines have a latency of 6 clock cycles; the latency parameters are provided by the Xilinx IP Core Generator and meet the requirements for a 125 MHz design frequency. Our design also supports a parameterized implementation at static time, where the FPU can be instantiated with a fused multiply-add (MADD) or divide unit. Since the FPU has only two read ports for VRF, the third operand in the multiply-add instruction is always a scalar supplied by one of the cores.

In our main implementation the Vector Processor contains 8 lanes and Vector Memory contains 8 low-order interleaved Xilinx BRAM banks with a total capacity of 64 Kbytes (8 banks x 8 Kbytes per bank). Without crossbar conflicts in accessing the Vector Memory banks, eight 32-bit data transfers can be performed on each clock cycle using the eight Load/Store units, giving a peak bandwidth of 32Gbs with a design frequency of 125 MHz. Of course, this bandwidth doubles with an expanded design for double-precision floating-point operations and transfers. Each BRAM is a true dual-port memory; one port is used for data transfers between the Vector Memory and the Vector Processor's register file, and the second port is used for data transfers between I/O controllers and the Vector Memory through the PLB interface. Therefore, this architecture supports concurrency and provides a high bandwidth for data transfers involving the Vector Memory.

2.1.3. Scheduling Procedure

The Scheduler controls the working context for the entire VP. Based on the chosen working state, the Scheduler provides configuration signals to all lanes and VCs. The signals for a particular lane provide information about: a) which VC the vector lane is assigned to, being VC 0, VC 1, or both; b) the total number of lanes assigned to the VC, including this particular lane; c) the offset/index of the lane in the lane array assigned to that VC; and d) the number of elements from a vector register which are located in this lane. The information from the first configuration signal (i.e., configuration a) is used by the lane to notify the appropriate VC of instruction completion, and the information derived from configurations b), c) and d) is used by the lane's LDST unit to properly translate addresses for memory accesses. The configuration signals provided to the VC by the Scheduler configure the former to work either in the exclusive context (i.e., one thread arriving from one scalar processor) or in the lane-sharing context (i.e., two distinct threads arriving from the two scalar processors).

```
STEP 1.1 Lock DMA resource
STEP 1.2 Transfer data from DDR to
         Vector Memory (VM)
STEP 1.3 Unlock DMA resource

STEP 2.1 Lock VP
STEP 2.2 Call FIR, FFT, Matrix Multiplication
         or LU Decomposition routines to
         process data from VM
STEP 2.3 Unlock VP

STEP 3.1 Lock DMA resource
STEP 3.2 Transfer processed data from VM to DDR
STEP 3.3 Unlock DMA resource
```

Figure 2. Main routine for CTS sharing

```
…
while (ack != IDLE) { //wait until the VP is idle
   LOCK ack;           // Scheduler returns a positive
                //or negative reply; 1-bit of information
}
  VLD VR0, A;   // Processor starts using the VP; loads
   //the vector register (A is address in Vector Memory)
  …
  VST VR4,  B;        // Processor finishes the routine;
                      // saves the vector result
                     // (B is address in Vector Memory)

UNLOCK ack;     // Unlock the VP resources and receives
               // a reply if successful or not;
…
```

Figure 3. CTS vector sharing routine

Each MicroBlaze can use a set of four indivisible instructions to communicate with the Scheduler. These are VP_REQ, VP_REL, VP_GETSTAT1 and VP_GETSTAT2; in response, the Scheduler always replies with a message. For a core to get access to the entire VP or to a subset of its lanes, the VP_REQ instruction is used. Based on the current VP state, any other pending VP requests, and the details of the current request, the Scheduler decides to grant a scalar processor request or not, and informs the requesting processor accordingly. In the extreme case where VP_REQ instructions arrive from both scalar processors in the same clock cycle, the Scheduler will reply to both of them but will positively acknowledge only one. For a successful request, the Scheduler will reply with the acquired Vector Length (VL) value and the acquired performance fields. In the case of an unsuccessful request, the Scheduler will transmit the available VL value and the currently available highest priority.

Table 1 shows some of the possible states for the Scheduler and state examples of each lane. In our current implementation, under CTS only one VC can issue an instruction to vector lanes at any time. In the FTS and VLS contexts, both VCs can issue simultaneously instructions to the lanes. The lane execution pipeline is capable of processing simultaneously instructions issued by both scalar processors since multiple vector instructions can simultaneously reside in the pipeline. Under these circumstances, FTS requires vector register renaming because the scalar processors must be assigned distinct vector registers. Usually small- and medium-scale SIMD machines are currently used as stream processors. Data can be streamed into the VM

of our VP-based structure using the DMA capability; the program then operates on this data using the VM as a data workspace, and the results are streamed back to the main memory using again DMA control. This data streaming can occur simultaneously with arithmetic computations. Figure 2 shows how the main routine of a MicroBlaze is developed for CTS/FTS and VLS sharing, and Figure 3 shows steps 2.1 to 2.3 for CTS sharing. Just before a thread becomes active, the software may clear all vector registers using a VP clear instruction. Another possibility is to implement additional hardware to support a local reset controlled by the Scheduler and triggered when the VP space is exclusively acquired by one of the scalar processors. When the scalar processor finishes the vector routine, it releases the coprocessor by issuing a VP_REL instruction. Prior to this instruction the MicroBlaze code makes sure that no vector register is dirty; also, the state of the vector processor for the respective MicroBlaze program is saved back into the memory. Therefore, the state of the VP must be saved before the VP is released in a shared environment.

Under FTS, vector instructions received from both scalar processors will share the VP resources. This context resembles fine-grain multithreading in superscalar processors, and increased throughput is expected because there are no data dependencies between instructions coming from different processors. More details about our VP architecture and Scheduler can be found in [20].

Table 1. Representative states of the Scheduler (left) and state examples of each lane (right). Each cell in the right side contains the state of the corresponding lane: which VC it is assigned to, the total number of lanes assigned to that VC, the lane index, and the number of elements from a vector register assigned to that lane; in state I VL(Vector Length)=32, state II VL=64; state III VL=64; state IV VL0=16; state V VL0=32/VL1=64.

	MicroBlaze 0	**MicroBlaze 1**
I	Coarse-grain sharing	No access
II	No access	Coarse-grain sharing
III	Fine-grain sharing	Fine-grain sharing
IV	Lane sharing (MB0 has access to M/2 lanes, i.e., half of the lanes)	No access
V	Lane sharing	Lane sharing

L0	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**
VC0 8/0/4	VC0 8/1/4	VC0 8/2/4	VC0 8/3/4	VC0 8/4/4	VC0 8/5/4	VC0 8/6/4	VC0 8/7/4
VC1 8/0/8	VC1 8/1/8	VC1 8/2/8	VC1 8/3/8	VC1 8/4/8	VC1 8/5/8	VC1 8/6/8	VC1 8/7/8
VC0/1 8/0/8	VC0/1 8/1/8	VC0/1 8/2/8	VC0/1 8/3/8	VC0/1 8/4/8	VC0/1 8/5/8	VC0/1 8/6/8	VC0/1 8/7/8
VC0 4/0/4	VC0 4/1/4	VC0 4/2/4	VC0 4/3/4	-	-	-	-
VC0 4/0/8	VC0 4/1/8	VC0 4/2/8	VC0 4/3/8	VC1 4/0/16	VC1 4/1/16	VC1 4/2/16	VC1 4/3/16

Table 2. Resource consumption in the Virtex XC5VLX110T FPGA device (8 Lanes and 8 Memeory Banks Configuration)

	Slice REGISTERs	**Slice LUTs**	**RAMB36_EXP**	**DSP48E**
LDST unit	713 (3.0%)	351 (2.1%)	-	2
ALU unit	1818 (7.7%)	1500 (8.9%)	-	2
VRF	258 (1%)	99 (<1%)	1	-
FVRF	44 (<1%)	44 (<1%)	-	-
LANE	2600 (11%)	1719 (10.3%)	1	4
VC	448 (1.9%)	296 (1.8%)	-	-
Scheduler	282 (1.2%)	357 (2.1%)	-	-
VP (8LANES)	21955 (92.9%)	14071 (84%)	8	32
VM	1820 (7.7%)	2874 (17.1 %)	16	-
VP+VM	**23628 (34%)**	**16765(24%)**	**16(16.2%)**	**32(50%)**

2.2. Resource Consumption and Resource Scalability

Table 2 shows resource consumption figures for our VP with 8 lanes and 8 memory banks configuration implemented in the Virtex XC5VLX100T FPGA device. Virtex-5 FPGAs contain a column-based architecture comprised of logic slices, 36-Kbit block RAMs called BRAMs (RAMB36_EXP), DSP slices (DSP48E) and many I/O hardwired IPs. Each logic slice can implement functions using four 6-input look up tables (LUTs) and four flip-flops; the LUTs can also be configured to realize dual-output 5-input LUTs. A LUT is a 64-bit memory capable of then realizing any of 32 or 64 functions. The DSP48E slice is based on a 25x18 bit multiplier and a 48-bit adder/subtractor/accumulator. Note that a vector lane contains an LDST unit, an ALU unit, a VRF and a Flag VRF (FVRF); the VP contains 8 lanes, 2 VCs and one Scheduler. Except for the last row in the table, the percentage values are shown relative to the total design resource consumption. As expected, most of the design is occupied by ALU units. Each lane consumes 1719 slice LUTs and 2600 slice registers (i.e., 10.3% and 11%, respectively, of the entire design), and the device consumption collectively by the VC and Scheduler is less than 4%. The overall device consumption by the VP and VM shown in the last row of Table 2 it is 16,765 slice LUTs and 23,628 slice registers, which represent 24% and 34%, respectively, of the VLX110T resources. The rest of the FPGA resources can be used for the realization of scalar processors, buses, DMAs, I/O controllers and other IPs.

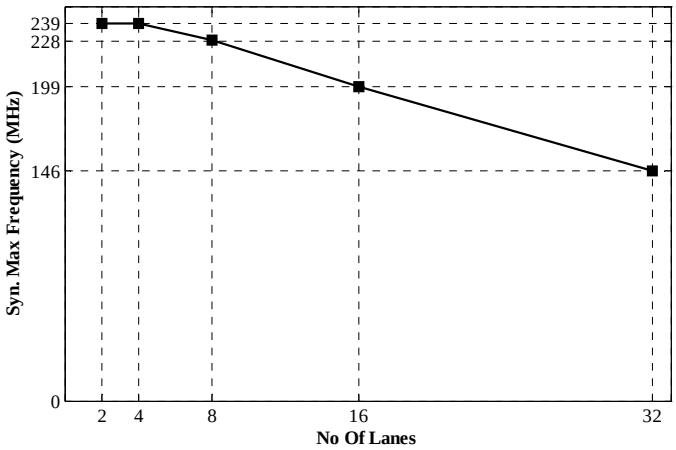

Figure 4. Maximum working frequency after synthesis for a Vector Processor with 2, 4, 8, 16 and 32 number of lanes on XC5VLX110T FPGA device. (Number of memory bank equals the number of lanes and the fully connected crossbar has size M×M).

Figure 4 displays the maximum frequency after synthesis for a VP design configured to have 2, 4, 8, 16 or 32 lanes. For the 2 and 4 lane configurations, the critical path lies in the vector lane logic; more explicitly, it involves the vector register file because this component runs at double the speed, i.e. 250 MHz. Starting with the 8-lane configuration, the crossbar becomes the timing bottleneck. For more than 16 lanes, other solutions for access to memory banks can be implemented in order to keep the working frequency high. The lane access to the memory banks follows the Uniform Memory Access (UMA) memory model and we can envision two solutions to scale the design: to implement a non-blocking multistage switch, and to adopt a Non-uniform Memory Access (NUMA) memory model.

3. Experimental Evaluation

3.1. Benchmarks

Five vector intensive programs, namely 32-tap FIR filtering, 32-point decimation-in-time radix-2 butterfly FFT, 1024x1024 dense matrix multiplication (MM), LU decomposition, and Sparse Matrix Vector Multiplication (SpMVM) were tested on our VP architecture. The performance evaluation numbers in our experiments include the I/O DMA transfer times (i.e., DDR to/from the VM). Our application coding process uses C macros exclusively to emit MicroBlaze custom instructions targeting the vector coprocessor, without modifying the gcc compiler system. The custom instructions (put, get, cput and cget) are communicated using Fast Simplex Link (FSL) channels. This method is similar to extending a core's ISA with custom instructions, but has the benefit of not making the overall speed of the core's pipeline dependent on these custom functions. Also, there are no additional requirements on the software tool chain associated with this type of functional extension. The system is simpler to program using macros. The routines for the VP were hand-coded, trying to improve the instruction throughput by using data prefetch via load instructions. In FTS and VLS sharing, there is no exclusive access to the VP so STEP 2.1 from Figure 4 is removed; that is, a request for VP resources can be granted without waiting for the VP to be idle. Each MicroBlaze uses its own partition in the VM and there are no data dependencies between threads running on the two processors. We can, however, envision scenarios where threads share data in the VM. Nevertheless, data sharing will not change the conclusions drawn in this paper about the benefits of coprocessor sharing for multicores in terms of performance and energy consumption. This is because multicores yield high performance when threads temporally running close to each other on different cores have limited or nonexistent inter- dependencies [21, 23]. In order to have exclusive access to the single DMA module, the Mutex IP core provided by Xilinx is used. The lock and unlock procedures for the DMA module require locking and unlocking the Mutex, respectively. For an in-depth evaluation of our architecture, for each benchmark we created several performance-power scenarios that involve loop unrolling, different vector lengths, instruction rearrangement optimization, and extensions of instruction set.

Table 3 summarizes the characteristics of the benchmark kernels that we used to test our sharing schemes. 32-tap FIR filtering is implemented using the outer product [17] that avoids the reduction operation. MM is based on the same procedure as FIR filtering and uses Single-precision real Alpha X Plus Y (SAXPY) in a loop to obtain one row result at the end of the loop. In some FIR and MM scenarios, pairs of multiply and add instructions are replaced by one fused multiply-add MADD instruction. FFT is implemented using a five-stage butterfly; each stage involves complex multiply and add vector operations, and a shuffle operation. LU decomposition consists of generating the L and U matrices from a dense 128×128 matrix using

the Doolittle algorithm. The algorithm starts with VL=128; when the number of nonzero elements in a row is less than half of the current VL value, the scalar processor changes the VL to half of the actual vector length for higher efficiency. Sparse Matrix Vector Multiplication (SpMVM) is implemented using the Compressed Row Storage (CSR) format and consists of two stages; under CSR, row traversal is applied for the storage of nonzero elements in sequential memory locations. In the first stage, the array values are multiplied with the corresponding elements from the vector; in the second stage, addition along each row is performed. The Load-Index (i.e., vector gather) instruction is intensively used in both stages. In order to speed-up the addition stage, the rows of the sparse matrix are stored in the increasing order of their number of non-zero elements.

Table 3. Summary of benchmark kernel characteristics.

Kernels	Scenarios obtained using combinations of	Implementation	Comments
32-tap FIR	CTS, FTS, VLS; VectorLength: VL={32, 64, 128, 256}; no loop unrolling, unrolled once and three times	Outer product	Some scenarios use the vector fused multiply-add (VMADD) instruction
32-point FFT	CTS, FTS, VLS; VL ={32, 64}; no loop unrolling, unrolled once	Five-stage butterfly	
Matrix Multi-plication MM (1024×1024)	CTS, FTS, VLS; VL ={32, 64, 128, 256}; no loop unrolling, unrolled once	Uses SAXPY	Some scenarios use the vector fused multiply-add (VMADD) instruction
LU decomposition	CTS, FTS, VLS; VL ={16, 32, 64, 128}; no loop unrolling	Doolittle algorithm on 128×128 matrix. When the number of nonzero elements in the row is less than half of the current VL value, the scalar processor changes the VL value to half of the actual vector length	Some scenarios use the vector division (VDIV) hardware support
SpMVM	CTS, FTS, VLS; VL={32}; no loop unrolling, unrolled once	Two stages: multiply and add.	Data is stored in the Compressed Row Storage (CSR) format. Load-Index (gather) instructions are used

For scenarios that utilize the VP continuously, the CTS context is equivalent to the case of having one multithreaded scalar processor that uses exclusively the VP. Compared to the classic implementation where a VP is always tied to the same scalar processor, the advantage of CTS in a multicore environment is that VP ownership can change dynamically for more robust application realization.

3.2. Evaluation Procedure

Execution times were obtained with ModelSim [22] simulations using the RTL system model. The Xilinx XPower tool [18] was then used to calculate the dynamic and static power dissipation based on data stored in the simulation record files (.vcd files recording the switching activities of all the logic and wires in the FPGA, and which are generated by ModelSim during the timing simulations with the place-and-route netlist). Static power measurements on the FPGA require adjustments to account for the fact that our VP configurations do not fully utilize the FPGA device. Accordingly, the static power consumption (also called quiescent power) reported by the Xilinx XPower tool for each type of fundamental FPGA block is scaled by the respective fraction of FPGA resources actually used by our design. To obtain realistic dynamic power figures, the timing simulations employed real floating-point input data. In all power calculations, all the design nets were matched; i.e., toggle information was extracted from all the nets in the netlist. Besides the execution times under various scenarios, we also produced figures for the average utilization of the ALU and LDST units. (per vector lane). The ALU average utilization is defined as the number of results produced by a lane's arithmetic and logic execution unit in 100 clock cycles, and the LDST utilization is the number of data words sent or received to or from the MC crossbar in 100 clock cycles.

3.3. Performance and Power Consumption Results

Tables 4, 5, 6, 7 and 8 show the ALU and LDST utilization, performance and power results in reference to the execution time for various configurations of the system: a) one scalar processor working without the VP and the DMA unit, and all data is pre-stored in the on-chip local memory; b) two scalar processors working without the VP and the DMA unit, and all data is pre-stored in the on-chip local memory; c) a scalar processor using exclusively the VP and the DMA unit (this represents CTS); d) two scalar processors working with the VP in the FTS context and the shared DMA unit; e) two scalar processors working with the VP in the VLS context and the shared DMA unit (each MicroBlaze acquires four lanes); and, for the sake of inclusiveness, f) the Intel Xeon general-purpose processor running at 3.2GHz in a commercial PC; the compilation was done using the O3 option for SSE3 instruction extensions. For each one of the c), d) and e) configurations, we present the results for some distinct scenarios that combine different vector lengths, loop unrolling and instruction extension. The VP configuration used to produce the performance and power results is M=8 lanes and L=8 memory banks. For FIR filtering, the

results are shown in ns per dot product. For FFT, the results are in µs per 32-point complex FFT operation, and for MM the results are in µs for the calculation of a single element in the product matrix. Besides the total execution time for the LU decomposition of a 128×128 dense matrix, Table 7 shows the time to process one single element for various vector lengths. Since recording a .vcd file for an entire LU decomposition task is impractical due to its size, Table 7 shows the power and energy dissipation for processing one row in Gaussian elimination. For SpMVM we used the *bccsstk13* matrix from Matrix Market [19], and the performance and energy results are presented per resulting vector.

Table 4. Performance and Power comparison for 32-tap FIR (8 Lanes and 8 Memory banks configuration)

		Average utilization (%)		Execution Time (ns)	Speedup	Dynamic Power (mW)	Energy (nJ)		nJ/FLOP
		ALU	LDST				Dynamic	Total	
One MB w/o VP		N/A	N/A	4060	1	-	-	-	-
Two MB w/o VP		N/A	N/A	2030	2	-	-	-	-
CTS	**VL=32; no loop unrolled**	**17.51**	**8.86**	**371.25**	**10.93**	**114.19**	**42.39**	**190.89**	**2.982**
	VL=32; no loop unrolled w/ MADD	11.99	11.99	275.5	14.74	-	-	-	-
	VL=128; no loop unrolled	**39.24**	**19.94**	**165.56**	**24.52**	**225.66**	**37.36**	**120.14**	**1.877**
	VL=128; no loop unrolled w/ MADD	28.06	28.06	117.62	34.51	-	-	-	-
	VL=128; unrolled three times	**83.31**	**42.51**	**78.31**	**51.85**	**479.28**	**37.53**	**68.85**	**1.075**
	VL=128; unrolled three times w/ MADD	84.96	72.22	45.94	88.38	-	-	-	-
FTS	**VL=32; no loop unrolled**	**34.97**	**17.70**	**186.01**	**21.83**	**220.98**	**41.10**	**115.50**	**1.804**
	VL=32; no loop unrolled w/ MADD	23.95	23.95	137.87	29.45	-	-	-	-
	VL=128; no loop unrolled	**75.66**	**38.24**	**85.98**	**47.22**	**432.74**	**37.21**	**71.61**	**1.118**
	VL=128; no loop unrolled w/ MADD	55.92	55.86	59.01	68.80	-	-	-	-
	VL=128; unrolled three times	**99.71**	**50.67**	**65.19**	**62.27**	**567.82**	**37.01**	**63.09**	**0.985**
	VL=128; unrolled three times w/ MADD	98.27	83.57	39.81	101.10	-	-	-	-
VLS	**VL=32; no loop unrolled**	**27.68**	**14.09**	**234.12**	**17.34**	**187.76**	**43.96**	**137.61**	**2.150**
	VL=32; no loop unrolled w/ MADD	19.39	19.29	170.25	170.25	-	-	-	-
	VL=128; no loop unrolled	**49.51**	**25.29**	**131.28**	**30.92**	**319.13**	**41.89**	**94.41**	**1.475**
	VL=128; no loop unrolled w/ MADD	35.91	35.91	91.78	44.23	-	-	-	-
	VL=128; unrolled three times	**89.89**	**45.71**	**72.21**	**56.22**	**554.97**	**40.07**	**68.96**	**1.077**
	VL=128; unrolled three times w/ MADD	91.43	76.97	42.69	95.104	-	-	-	-
GPP Xeon		-	-	340.08	11.94	N/A	-	-	-

Table 5. Performance and Power comparison for 32-point Complex FFT (8 Lanes and 8 Memory banks configuration)

		Average utilization (%)		Execution Time (µs)	Speedup	Dynamic Power (mW)	Energy (nJ)		nJ/FLOP
		ALU	LDST				Dynamic	Total	
One MB w/o VP		-	-	160.01	1	-	-	-	-
Two MB w/o VP		-	-	80.01	2	-	-	-	-
CTS	**VL=32; no loop unrolled**	**43.29**	**23.38**	**3.264**	**49.02**	**233.59**	**762.40**	**2068.01**	**3.231**
	VL=32; unrolled once	65.10	34.78	2.172	73.66	330.07	716.91	1585.71	2.477
	VL=64; unrolled once	**78.92**	**43.09**	**1.782**	**89.78**	**398.79**	**710.64**	**1423.44**	**2.224**
FTS	VL=32; no loop unrolled	76.28	42.39	1.844	86.76	405.14	747.07	1484.46	2.319
	VL=32; unrolled once	**87.20**	**46.44**	**1.618**	**98.89**	**456.32**	**738.32**	**1385.52**	**2.164**
	VL=64; unrolled once	89.45	48.60	1.573	101.72	460.97	725.11	1352.72	2.113
VLS	**VL=32; no loop unrolled**	**62.74**	**35.11**	**2.192**	**72.99**	**356.43**	**781.29**	**1658.09**	**2.590**
	VL=32; unrolled once	74.23	41.60	1.848	86.58	406.09	750.45	1489.65	2.327
	VL=64; unrolled once	**79.18**	**44.56**	**1.701**	**94.06**	**429.24**	**730.14**	**1410.53**	**2.203**
GPP Xeon		-	-	100.01	1.60	-	-	-	-

Table 6. Performance and Power comparison for Matrix Multiplication (8 Lanes and 8 Memory banks configuration)

		Average utilization (%)		Execution Time (µs)	Speedup	Dynamic Power (mW)	Energy (nJ)		nJ/FLOP
		ALU	LDST				Dynamic	Energy (nJ)	
One MB w/o VP		-	-	130.90	1	-	-	-	-
Two MB w/o VP		-	-	65.45	2	-	-	-	-
CTS	**VL=32; no loop unrolled**	**20.37**	**20.70**	**10.09**	**12.97**	**166.22**	**1677.16**	**5713.16**	**2.789**
	VL=32; no loop unrolled w/ MADD	12.65	25.11	8.73	14.99	-	-	-	-
	VL=32; unrolled once	**33.94**	**34.50**	**6.03**	**21.71**	**296.69**	**1787.85**	**4198.25**	**2.049**
	VL=32; unrolled once w/ MADD	17.78	36.20	5.74	22.80	-	-	-	-
	VL=128; unrolled once	**68.30**	**69.51**	**3.01**	**43.49**	**555.02**	**1671.16**	**2875.68**	**1.404**
	VL=128; unrolled once w/ MADD	35.00	72.04	2.912	44.95	-	-	-	-
FTS	**VL=32; no loop unrolled**	**40.59**	**41.29**	**5.055**	**25.89**	**332.86**	**1682.61**	**3704.60**	**1.808**
	VL=32; no loop unrolled w/ MADD	16.51	43.12	4.981	26.28	-	-	-	-
	VL=32; unrolled once	**67.09**	**68.20**	**3.048**	**42.95**	**610.20**	**1859.89**	**3079.08**	**1.503**
	VL=32; unrolled once w/ MADD	34.01	72.06	2.882	45.42	-	-	-	-
	VL=128; unrolled once	**97.32**	**98.91**	**2.102**	**62.27**	**793.65**	**1668.25**	**2509.05**	**1.225**
	VL=128; unrolled once w/ MADD	48.94	99.03	2.091	62.17	-	-	-	-
VLS	**VL=32; no loop unrolled**	**33.83**	**34.34**	**6.086**	**21.51**	**311.86**	**1897.98**	**4332.38**	**2.115**
	VL=32; no loop unrolled w/ MADD	18.58	37.12	5.883	22.09	-	-	-	-
	VL=32; unrolled once	**53.51**	**54.45**	**3.791**	**34.53**	**513.59**	**1947.02**	**3463.42**	**1.691**
	VL=32; unrolled once w/ MADD	27.46	55.64	3.692	35.45	-	-	-	-
	VL=128; unrolled once	**81.88**	**83.40**	**2.494**	**52.48**	**668.76**	**1667.89**	**2665.49**	**1.301**
	VL=128; unrolled once w/ MADD	41.59	83.97	2.490	52.57	-	-	-	-

GPP Xeon	-	-	20.56	6.36	-	-	-	-

Table 7. Performance and Power comparison for LU decomposition (8 Lanes and 8 Memory banks configuration)

		Average utilization (%)		Average Execution Time per element (ns)	Execution Time (µs) for entire LU dec.	Speedup	Dynamic Power (mW)	Energy (nJ)		nJ/FLOP
		ALU	LDST					Dynamic	Energy (nJ)	
One MB w/o VP		N/A	N/A	N/A	1,034,340	1	-	-	-	-
Two MB w/o VP		N/A	N/A	N/A	517,170	2	-	-	-	-
CTS	**VL=16**	**4.73**	**5.34**	**39.50**	5,137	201	46.03	29.09	281.89	8.809
	VL=32	**9.88**	**10.36**	**19.75**			85.46	54.01	306.81	4.794
	VL=64	**20.11**	**20.42**	**9.86**			164.71	104.09	356.89	2.788
	VL=128	**40.44**	**40.54**	**4.94**			317.69	200.78	453.58	1.771
CTS w/ VDIV	VL=16	8.38	8.43	17.75	4,213	245	-	-	-	-
	VL=32	16.10	16.22	10.00			-	-	-	-
	VL=64	31.12	31.58	6.00			-	-	-	-
	VL=128	44.06	44.09	4.44			-	-	-	-
FTS	**VL=16**	**8.32**	**8.54**	**20.06**	2,568	402	87.24	27.21	152.01	4.750
	VL=32	**18.74**	**21.08**	**9.88**			132.95	42.01	168.41	2.631
	VL=64	**39.93**	**41.36**	**4.94**			252.94	79.92	206.32	1.611
	VL=128	**81.05**	**82.30**	**2.47**			471.15	148.88	275.28	1.075
FTS w/ VDIV	VL=16	16.12	16.34	8.94	2,107	490	-	-	-	-
	VL=32	31.34	31.89	5.00			-	-	-	-
	VL=64	59.63	60.38	3.00			-	-	-	-
	VL=128	85.24	85.65	2.22			-	-	-	-
VLS	**VL=16**	**8.70**	**11.11**	**20.00**	3,522	293	89.82	28.74	156.74	4.898
	VL=32	**19.05**	**21.03**	**9.88**			157.59	49.79	176.19	8.809
	VL=64	**39.62**	**41.05**	**4.94**			290.38	91.76	218.16	4.794
	VL=128	**53.86**	**54.95**	**3.69**			422.28	199.31	388.11	2.788
VLS w/ VDIV	VL=16	13.11	13.89	10.440.167	2,380	434	-	-	-	-
	VL=32	27.08	27.73	5.78			-	-	-	-
	VL=64	50.11	50.87	3.36			-	-	-	-
	VL=128	75.87	75.98	2.51			-	-	-	-
GPP Xeon		-	-	-	89,060	11.62	-	-	-	-

Table 8. Performance and Power comparison for Sparse Matrix Vector Multiplication (8 Lanes and 8 Memory banks configuration); sparse matrix is *bcsstk13*.

		Average utilization (%)		Execution Time (µs)	Speedup	Dynamic Power (mW)	Energy (nJ) / vector result		nJ/FLOP
		ALU	LDST				Dynamic	Energy (nJ)	
One MB w/o VP		-	-	59,018	1	-	-	-	-
Two MB w/o VP		-	-	29,509	2	-	-	-	-
CTS	VL=32	20.90	9.52	2,877	20.51	134.55	387,100	1,537,900	9.167
FTS	VL=32	39.39	19.92	1,711	34.49	229.82	393,222	1,077,622	6.423
VLS	VL=32	33.11	16.79	2,020	29.22	210.63	425,472	1,233,472	7.352
GPP Xeon		-	-	8,401	7.025	-	-	-	-

From these performance results the following conclusions can be made: i) the best performance is provided by FTS followed by VLS and CTS; ii) although we have not included for simplicity the results for all possible vector lengths, our results show that a higher VL value increases the data-level parallelism, and therefore the performance; iii) loop unrolling increases the utilization of the units and also the overall performance; iv) with a low utilization of the units the speedup doubles from CTS to FTS (see VL=32 without loop unrolling for FIR, FFT, MM and LU); moreover, if the utilization from each thread is less than 50%, the speedup of FTS almost doubles as compared to CTS; v) for kernels with a high utilization of the lane units in the CTS mode, FTS can provide a speedup of 1.2 to 1.5 as compared to CTS. This is caused by the fact that FTS achieves close to 100% utilization (peak performance) and the VP can no longer accommodate more instructions in its pipeline; vi) thread-level parallelism can provide higher speedup than data-level parallelism and loop unrolling (for FFT, FTS with VL=32 and without loop unrolling yields almost the same performance as CTS with VL=64 and the loop unrolled once). Therefore, we can overcome the lack of data-level parallelism and inadequate compiler optimization (loop unrolling) for an application by simultaneously processing an additional thread. LU decomposition exhibits low utilization for low vector lengths, especially when the division is done by MicroBlaze. This is caused by the scalar MicroBlaze that involves one floating-point division and two memory accesses per processed row; it can fully overlap VP code runs. As a consequence, two scalar processors in the FTS context provide a speedup of two as compared to the CTS context. This is a good example of applications where the fraction of sequential code is substantial and the utilization of the VP accelerator is low. Thus, adding threads from two or more processors will increase the speedup almost linearly for the same VP resources. As compared to Xeon, FTS provides a speed-up between 5 (for FIR) and 63 (for FFT) despite the much lower operating frequency of our FPGA-based prototype.

Under real application scenarios, requests initiated by the cores for VP processing are separated by periods of time during which the cores are doing scalar operations or stay idle. This may occur during the execution of a single or multiple threads running on a single core with a dedicated VP, or during the execution of threads running on multiple cores sharing a VP. Accordingly, we have produced relevant execution scenarios. Table 9 shows the normalized execution time for two threads composed of vector kernels chosen randomly from a set of ten kernels, where two consecutive kernels in a thread are separated by an idle period of random duration. The average ratio idle/busy of VP idle and busy times in Table 9 is defined as the ratio between the average duration of idle periods and the average time that a kernel utilizes the VP. Without idle periods (i.e., the ratio is zero), CTS yields very similar performance with a single core having a dedicated VP with the same total number of lanes (eight in our executions); FTS gives the best performance and is followed by VLS. As the ratio increases, the performance gap between CTS and a single core with a dedicated VP increases further, in favor of CTS. This shows once more the effectiveness of VP sharing compared to the case of a dedicated VP per core.

Table 9. Normalized execution time relative to a CPU with a dedicated 8-lane VP and no idle periods.

	Average ratio of VP idle/busy times (%)				
	0	**25**	**50**	**75**	**100**
1 CPU with 8 lanes in a dedicated VP	1	1.23648	1.55319	1.71154	1.79072
CTS with a VP (8 lanes shared by 2 CPUs)	0.99839	1.00094	1.05684	1.10314	1.12897
VLS (4 distinct VP lanes per CPU)	0.66259	0.78084	0.93919	1.01837	1.05796
FTS (8 lanes shared by 2 CPUs)	0.55194	0.65907	0.80844	0.88438	0.92157

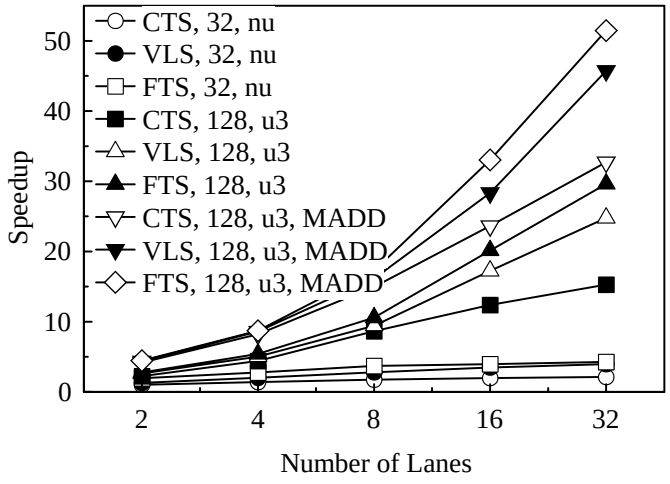

Figure 5. FIR routine for 2, 4, 8, 16 and 32 lane configurations. Each application involves sharing context, Vector Length, unroll type (nu=no unroll; u3=unrolled three times), and with or without VMADD instruction extension.

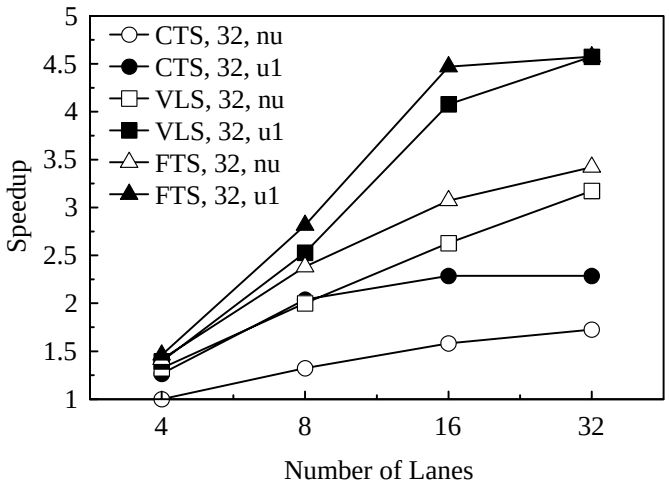

Figure 6. FFT routine for 4, 8, 16 and 32 lane configurations. Each application involves sharing context, Vector Length, and unroll type (nu=no unroll; u1=unrolled once).

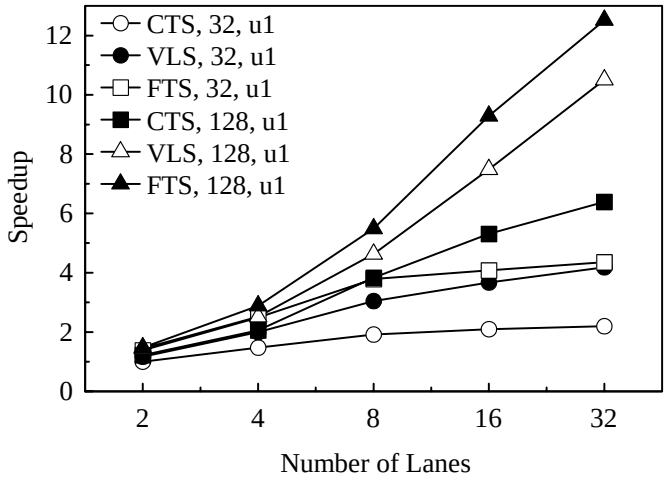

Figure 7. MM routine for 2, 4, 8, 16 and 32 lane configurations. Each application involves sharing context, Vector Length, and unroll type (u1=unrolled once).

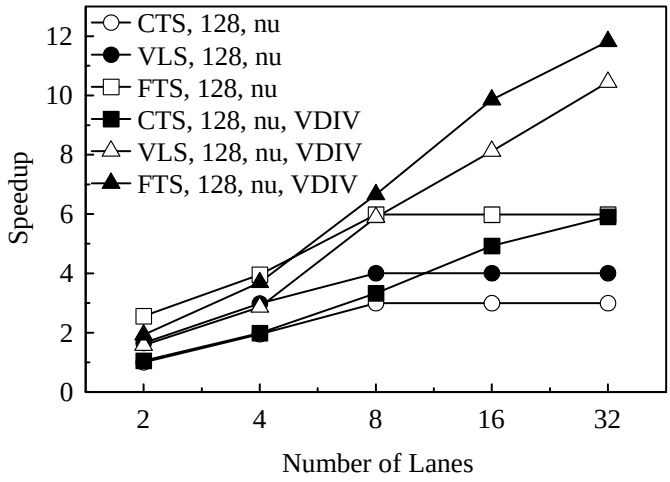

Figure 8. LU decomposition routine for 2, 4, 8, 16 and 32 lane configurations. Each application involves sharing context, Vector Length, and unroll type (nu=no unroll), and with or without VDIV instruction extension.

In FPGAs the static power consumption is dominated by the leakage current in the transistors. Leakage current occurs across gates of transistors (through thin oxides) and sub-threshold leakage from drain to source. The dominant cause of dynamic power consumption is the charging and discharging of capacitance within the device as it manipulates or moves data during computation. The power tables show that: i) the lowest dynamic energy is provided by FTS followed by CTS and VLS, with the values having a small dispersion; ii) however, if static power is included, the advantage of FTS and VLS is substantial, especially for low average utilization (see the FIR benchmark for CTS, FTS, and VLS with VL=32 and no unrolling); iii) adding a new thread has almost the same performance gain and total energy consumption as doubling the data-level parallelism and unrolling the loop once (see FFT under CTS with VL=64 and loop unrolled once as compared to FFT under FTS with VL=32 and without loop unrolling). Under similar LDST utilization, the MC crossbar and VM dynamic power consumption is higher in VLS than in any other VP sharing context. This is due to the high contention in the crossbar due to the presence of two LDST threads corresponding to two distinct VPs, with no synchronization for accessing the Crossbar Memory.

3.4. Performance Scalability

In order to analyze the scalability of the proposed VP-sharing schemes, we benchmarked four of the applications for VP configurations with 2, 4, 8, 16 and 32 lanes. Figure 5. to 8 show that the FTS scheme most often scales better than CTS and VLS for large vector lengths. Also, the application scales better with increasing data parallelism caused by high vector length and loop unrolling. For the FIR application, the fused multiply-add MADD instruction extension increases the speedup with almost 60% compared to the corresponding schemes without MADD. In LU decomposition, all schemes without the VDIV extension provide the same performance with 8, 16 and 32 lanes in the configuration. This is caused by the scalar code that runs on MicroBlaze, involves one floating-point division and two memory accesses per processed row, and fully overlaps VP runs. As a consequence, two scalar processors under FTS provide a speedup of two as compared to CTS. This is a good example of applications where the fraction of sequential code is substantial and the utilization of the VP accelerator is low. However, the inclusion of division in the FPUs allows the offloading of the scalar processors, thus improving the performance as the number of lanes increases. It can be observed that FTS with VDIV provides almost 100% improvements in the 32-lane configuration as compared to FTS without the VDIV extension. It should be also emphasized that for applications with low parallelism increasing the number of lanes does not improve the performance.

4. Conclusions

In this paper we presented three schemes for improving the throughput of a shared vector coprocessor in a multicore environment. Coarse-grain temporal sharing (CTS) consists of temporally multiplexing sequences of vector instructions ideally arriving from different threads. However, providing a per core exclusive access to the vector resources does not maximize their utilization. Fine-grain temporal sharing (FTS) consists of spatially multiplexing individual instructions issued by different scalar processors, in order to increase the utilization of the functional units. Finally, vector-lane sharing (VLS) consists of simultaneously allocating distinct vector lanes or collections of them to distinct cores. We evaluated the

performance and energy consumption for these coprocessor sharing contexts by implementing several floating-point applications on an FPGA-based prototype. FTS exhibits the biggest speedup and smallest energy consumption, and is followed by VLS. Moreover, under low resource utilization FTS doubles the speed-up and reduces the energy consumption by 50 percentages as compared to the case where a core has exclusive access to the vector coprocessor.

These models suggest several techniques to increase the performance or reduce the energy consumption: i) increase the data-level parallelism by increasing the vector length; ii) increase the instruction-level parallelism at compile time by loop unrolling or other techniques; iii) use multiple threads in a multiprocessor environment to increase the vector coprocessor utilization. Our analysis showed that the last technique can be superior to the former two combined. Therefore, the lack of adequate data-level parallelism in an application can be overcome by sharing the coprocessor resources among many cores.

Also, we extended the FPU by adding a fused multiply-add MADD unit or a divider. Our results show that the speedup improves almost by 60% for FIR and by 100% for LU decomposition. Finally, we analyzed the scalability of our VP sharing schemes for various numbers of vector lanes. FTS scales the best among all presented schemes, especially when the applications are implemented using a high vector length and loop unrolling.

Future work will focus on providing Quality-of-Service (i.e., the VC and Scheduler will assign coprocessor resources based on the priorities of the active threads). Preemptive coprocessor scheduling will be investigated and this sharing approach will be extended to coprocessors of other type. Power scalability will be investigated also in a future work. Finally, since the single precision floating-point MADD unit occupies almost the same resources as the divide and the square root units, they could be integrated in a runtime reconfigurable module.

5. References

[1] C. Kozyrakis and D. Patterson, “Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks,” 35th Annual IEEE/ACM International Symposium on Microarchitecture, 283–293, 2002.

[2] C. Kozyrakis and D. Patterson, “Overcoming the limitations of conventional vector processors,” SIGARCH Comput. Archit. News, 31(2):399–409, 2003.

[3] C. Kozyrakis and D. Patterson, “Scalable, vector processors for embedded systems,” IEEE Micro, 23(6):36–45, 2003.

[4] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, “AnySP: Anytime Anywhere Anyway Signal Processing,” IEEE Micro, 30(1), 81-91, Jan./Feb. 2010

[5] http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf, Xilinx Inc., MicroBlaze Processor Reference Guide, 2008.

[6] J. Cho, H. Chang, and W. Sung, “An FPGA based SIMD processor with a vector memory unit,” IEEE International Symposium on Circuits and Systems, pp. 525-528, 2006.

[7] Y. Lin, et al. “SODA: A low-power architecture for software radio,” 33rd Annual International Symposium on Computer Architecture, pages 89-101, 2006.

[8] P. Yiannacouras, J. G. Steffan, and J. Rose, “VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors,” International Conference on Compilers, Architecture and Synthesis for Embedded Systems, October 2008, Atlanta, GA.

[9] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, “Vector Processing as a Soft Processor Accelerator,” ACM Transactions on Reconfigurable Technology and Systems, Volume 2, Issue 2, June 2009.

[10] H. Yang and S. Ziavras, “FPGA-Based Vector Processor for Algebraic Equation Solvers,” IEEE International Systems-On-Chip Conference, Washington, D.C., Sept. 25-28, 2005.

[11] A. Azevedo and B. Juurlink, “Scalar Processing Overhead on SIMD-Only Architectures,” 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, pp.183-190, 7-9 July 2009

[12] F. Gerneth, “FIR Filter Algorithm Implementation using Intel SSE instructions: Optimizing for Intel Atom architecture,” Software White Paper on Intel Embedded Design Center, March 2010 (http://download.intel.com/design/intarch/papers/323411.pdf).

[13] F. Sanchez, M. Alvarez, E. Salami, A. Ramirez, M. Valero, "On the Scalability of 1- and 2-Dimensional SIMD Extensions for Multimedia Applications," IEEE International Symposium on Performance Analysis of Systems and Software, pp. 167-176, ISPASS 2005.

[14] M. Keating, D. Flynn, R. Aitken, A. Gibsons and K. Shi, “Low Power Methodology Manual for System on Chip Design”, Springer Publications, NewYork, 2007.

[15] S.F. Beldianu and S.G. Ziavras, “On-chip vector coprocessor sharing for multicores,” 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, IEEE Computer Society Proceedings, PDP 2011, Ayia Napa, Cyprus, pp. 431-438, 9-11 February 2011.

[16] C.E. LaForest and J. G. Steffan, “Efficient Multi-Ported Memories for FPGAs,” 18th annual ACM/SIGDA international symposium on Field Programmable Gate Arrays, ACM, New York, NY, USA, 41-50.

[17] W. Sung and S.K. Mitra, “Implementation of digital filtering algorithms using pipelined vector processors,” Proceedings of the IEEE , vol.75, no.9, pp. 1293- 1303, Sept. 1987.

[18] “XPower Estimator User Guide,” Xilinx, www.xilinx.com/support/documentation/user_guides.

[19] http://math.nist.gov/MatrixMarket/.

[20] S. F. Beldianu, and S. G. Ziavras, “Multicore-based vector coprocessor sharing for performance and energy gains,” ACM Transactions on Embedded Computing Systems, accepted for publication.

[21] M. A. Hossain, U. Kabir and M. O. Tokhi, “Impact of data dependencies for real-time high performance computing”, Journal of Microprocessors and Microsystems, 26(6), 253 –261, 2002.

[22] http://www.actel.com/documents/modelsim_ug.pdf, ModelSim User’s Manual, Mentor Graphics Corp. , 2010.

[23] Y. Wang, et al., “Investigation of Factors Impacting Thread-Level Parallelism from Desktop, Multimedia and HPC Applications,” Proceedings of the 4th International Conference on Frontier of Computer Science and Technology, 17-19 Dec., 2009, Shanghai, China, 27-32.