Vector Coprocessor Virtualization for Simultaneous Multithreading

YAOJIE LU, SEYEDAMIN ROOHOLAMIN and SOTIRIOS G. ZIAVRAS,
New Jersey Institute of Technology, Dept. of Electrical and Computer Engineering

Vector coprocessors (VPs) often are not used efficiently due to data dependencies between instructions and limited portion of vectorizable code in single applications. We present a VP sharing technique for multithreaded processors or multicores where a VP can simultaneously execute multiple threads of similar or disparate vector lengths for improving VP throughput. A vector register file (VRF) virtualization technique is invented which dynamically allocates physical vector registers to threads. Virtualization improves programmer productivity by providing at run time a fixed register name space to each competing thread, thus eliminating the need to solve register name conflicts statically. We prototyped our virtualization technique for a multi-core processor embedded in an FPGA and evaluated its performance for various benchmarks. Under the dynamic creation of disparate threads, the results show impressive VP speedups of up to 333% and total energy savings of up to 37% with proper thread scheduling and power gating compared to a similar-sized system that allows VP access to just one thread at a time.

• Computer systems organization~Single instruction, multiple data • Computer systems organization~Multicore architectures

General Terms: Performance, Power.

Additional Key Words and Phrases: Vector coprocessor, virtualization, coprocessor sharing, multicore, FPGA prototyping, energy savings.

1. INTRODUCTION

SIMD architectures are highly efficient in exploiting data level parallelism (DLP) in applications due to their specialization. A VP, also known as array processor, employs an SIMD architecture capable of processing an array of data elements simultaneously by executing a single vector instruction. As an accelerator, a VP can offload the DLP workload from general-purpose processors, thus enhancing the overall performance and energy efficiency. The VIRAM's multi-lane architecture is the basis of several VP designs [Kozyrakis and Patterson 2003]. VIRAM has separate pipeline structures for load-store (LDST) units and arithmetic logic units (ALUs). Vector registers are distributed evenly across the vector lanes. Each lane carries out ALU array operations on data within its local VRF. Vector elements in a lane are processed sequentially due to the ALU's pipelined architecture while all lanes work in parallel on different array parts. SODA [Lin et al. 2006] is a fully programmable VP that realizes the W-CDMA and IEEE802.11a protocols. Lee et al. [2013] compared accelerators having MIMD (Multiple-Instruction Multiple-Data), vector SIMD and vector thread (VT) architectures in reference to programmability and implementation efficiency; it confirmed that SIMD vector architectures exploit DLP more efficiently than MIMD even for irregular data pattern accesses.

Multicores with embedded VPs have been implemented on FPGAs. They are often referred to as soft vector processors (SVPs). Cho et al. [2006] introduced an SVP that eliminates memory bank conflicts by using address generation and rearrangement units in the vector memory. VESPA [Yiannacouras et al. 2008] allows the addition of vector lanes with minimum hardware modifications. Benchmarking shows a speedup of 6.3 under saturation with 16 lanes compared to a single lane. VIPERS achieves a speedup of 44 compared to the Nios II scalar processor [Yu et al. 2009]. Its number of functional units and register file bandwidth are configured by software; it occupies 25 times more area compared to its host scalar core. VEGAS [Chou et al. 2011] improves VIPERS's area-delay product. Using scratchpad memory instead of a VRF, it achieves a speedup of up to 208 compared to Nios II. Further improvements to reduce ALU bottlenecks produced VENICE [Severance and Lemieux 2012] that doubles the performance-per-logic block compared to VEGAS. With the integration of a streaming pipeline in the data path, a speedup of 7000 over a scalar processor was reported for the N-body problem [Severance et al. 2014].

Application specific VPs are optimized for certain applications. Multimedia applications containing video processing kernels deal with massive DLP. SIMD vector architectures are the best candidates to exploit the parallelism in video frames. Many researchers have tried to optimize codecs for the implementation of new video coding standards such as H.264 or MPEG4. Iranpour and Kuchcinski [2004], Lee et al. [2004], Kim et al. [2005], Shengfa et al. [2006], and Lee et al. [2009] propose SIMD-based video codecs. A major challenge in video applications is irregular data accesses for video compression. Lo et al. [2011] overcome this issue by inserting a crossbar between the ALUs and the VRF. Yang et al. [2005] propose an application-specific VP prototyped on an FPGA for sparse matrix multiplication. IBM's PowerEN processor integrates five hardwired application-specific accelerators in a heterogeneous architecture for key functions such as compression, encryption, authentication, intrusion detection and XML processing. This approach facilitates energy-proportional performance scaling [Heil et al. 2014].

2. MOTIVATION AND RELATED WORK

Our work is motivated by the fact that single-thread dedicated VPs are often not efficiently utilized for the following reasons. First, every application contains some unvectorizable serial code for flow control or other system management; the scalar host may not issue vector instructions to the VP at a rate sufficient to keep it highly utilized. Second, data dependencies within some applications' vector instruction flows can cause frequent stalls, wasting precious clock cycles in the VP's super-pipelined floating-point units (FPUs). Finally, it may be preferable that applications with small vectorizable code be executed on the scalar host in order to give another highly-vectorized application exclusive VP access. However, the former applications as well could benefit from using simultaneously the VP. Our benchmarking shows that applications with VP utilization as low as 8.5% can yield a speedup of 84 by executing on a VP compared to a scalar processor with the same clock frequency.

Traditional VPs designed to service exclusively one host scalar processor are normally optimized for applications of a certain level of DLP and usually more vector lanes can be added to exploit other applications of higher DLP [Kozyrakis and Patterson 2003; Yiannacouras et al. 2008; Yu et al. 2009]. However, an increased number of lanes will reduce the VP utilization for the lower-DLP applications.

To address these challenges, we propose a VP sharing technique for simultaneous multithreading that achieves high aggregate VP utilization independent of the DLP of individual vector threads. VP sharing for multiple threads or cores was first proposed by Beldianu and Ziavras [2013]. Three sharing policies were introduced for a multi-lane VP, namely coarse-grain temporal sharing (CTS), fine-grain temporal sharing (FTS) and vector lane sharing (VLS). Their FPGA prototype contained two scalar core processors. Under CTS, each core reserves the entire VP exclusively until its current vector thread stalls or completes execution, and then hands over VP access to the other core. FTS is similar to the VP sharing scheme we propose here, where all cores access the entire VP simultaneously and VP resource conflicts are resolved by an arbitrator. CTS and FTS support sharing only for threads with the same VL (vector length; it represents the number of elements in the vector). VLS is the only mode under which active threads using different VLs can coexist in the VP which is split into two independent sets of vector lanes, one set for each core; VLS relies on two vector controllers (VCs) to control the two sets. FTS achieves the best VP utilization and may double the speedup compared to CTS while reducing the dynamic energy by 50% [Beldianu and Ziavras 2014].

Our VP sharing technique is similar to Intel's proprietary Hyper-Threading Technology (HTT), a simultaneous multithreading technique for general-purpose processors that "makes a single physical processor appear as multiple logic processors" [Marr et al. 2002]. In contrast, our technique applies simultaneous multithreading to vector code. Another difference is that threads may arrive from different cores. A third difference is that each logic processor in HTT contains a complete set of general-purpose registers due to a limited register space; due to 2D VRF configuration, however, a similar VP approach with a unique VRF for each vector thread would require substantial resources with expected small utilization. Our VRF virtualization technique addresses this issue by maintaining a unique logical VRF space per thread while all threads share a physical VRF.

To improve the performance of Beldianu and Ziavras [2013], we introduced in Rooholamin and Ziavras [2015] a multi-lane VP with separate pipelines for the ALU and LDST units. A lane's LDST unit exclusively accesses a port of a distinct vector memory (VM) bank, thus eliminating contentions that can lead to delays or stalls for sequential read/write operations. All non-sequential memory accesses (e.g., data shuffling and index addressing) are handled by per-lane dedicated shuffle engines that utilize a second port of each VM bank. This VP is highly flexible for applications with varying VL, thus allowing the VL value to be specified by each individual vector instruction; the instruction decoder in each lane is then in charge of vector instruction synchronization. Threads of disparate VLs running on the same scalar processor can exploit the VP in a CTS-like fashion as long as they do not result in vector register name conflicts. Benchmarking showed speedups of up to 1500 compared to running vector code on a scalar processor with the same clock frequency.

For our current paper, we first extend the VP prototype in Rooholamin and Ziavras [2015] to include three additional scalar core processors, for a total of five scalar cores. Some minor hardware modifications are carried out as well. Four cores share the VP simultaneously for running vector applications whereas the fifth core does VP management and vector thread scheduling. The new VP can run simultaneously up to four vector threads with different VL. Any vector register name conflicts between threads are resolved via an innovative VRF virtualization technique. Virtualization involves an effective register management algorithm run on the control core and a hardwired translation look-up table (TLT) for fast virtual to physical register name (i.e., ID) translation. With VRF virtualization, the management of physical vector register names becomes transparent to application programmers who assume a virtual register space. Our benchmarking shows a throughput improvement of up to 400% for many low VP utilization applications compared to the older VP that did not support simultaneous multithreading. We also propose here a high throughput runtime scheduler for VP threads. Our experiments show 322% throughput improvement and energy savings of 37% with proper scheduling and power-gating that reduces static energy.

Our work also differs from Beldianu and Ziavras [2013] in other aspects. Most importantly, our VRF virtualization greatly improves in practice simultaneous VP sharing. Otherwise, application programmers must rename vector registers statically based on thread combinations that will be present simultaneously in the VP; this is hardly possible in dynamic environments with an unknown, large or infinite number of combinations. Second, Beldianu and Ziavras [2013] supports VP sharing for two threads of different VL only under the VLS execution mode that configures two independent sets of vector lanes using two VCs. In contrast, we maximize the VP's utilization by allowing multiple threads of different VL to run simultaneously on the VP. This results in substantial throughput increases. A single VC broadcasts vector instructions to all lanes. The thread ID and VL reside in each broadcasted instruction. With multiple non-empty instruction FIFOs, round-robin arbitration decides each clock cycle the vector instruction to enter the VP. The thread population in the VP can be increased by modifying the arbitrator's state machine. Third, an added FIFO structure between the VP interface and host cores eliminates frequent stalls of the latter due to vector instruction arbitration. Under low VP utilization, an application's speed is bounded by its host core. However, when multiple hosts send simultaneously vector instructions to the VP, then only one host will get VP access in the next clock cycle. Such wastage of clock cycles can be avoided with the implemented FIFOs since a core can keep sending vector instructions until its FIFO becomes full; this will occur for peak VP utilization. Finally, we removed the crossbar between lanes and VM banks by connecting a bank's dedicated port to the attached lane's LDST unit. This modification eliminates arbitrator delays in the crossbar and improves VP throughput for sequential memory accesses that are omnipresent due to its pipelined units that target array operations.

The architecture of the scalar processors sub-system and its VP interface are covered in Section 3. The VRF virtualization technique is introduced in Section 4. The VP architecture is discussed in Section 5. The resource consumption of our FPGA prototype is covered in Section 6. Section 7 introduces benchmarks and performance results. Section 8 proposes a throughput-oriented scheduler. Section 9 analyzes the VP's power and energy consumption. Finally, conclusions are drawn in Section 10.

3. THE MULTI-CORE ARCHITECTURE

As shown in Fig. 1, our prototype consists of two sub-systems: the scalar processors sub-system (SPS) with five cores and the VP. SPS does system management and runs flow control parts of application code; it sends vector instructions to the VP. TLT provides hardware support for real-time VP register renaming; it is managed by SPS. AXI4 (Advanced eXtensible Interface 4.0) interconnects SPS components. Two AXI4 types, AXI4-Stream (AXI4-S) and AXI4, are also present. AXI4-S pairs realize bidirectional handshaking. The interface between SPS and VP is pipelined, and VP can read up to one 32-bit instruction/datum and three 6-bit physical register names from SPS per clock cycle.

MicroBlaze (MB), a Xilinx 32-bit RISC soft processor, forms SPS cores MB0-MB4. Its Harvard architecture interfaces a fast local memory (LM) in in Fig. 1; LM contains frequently used library functions. LM blocks are initialized from the FPGA's flash memory upon power up; these connections are omitted. The libraries can be modified at runtime by MBs. In addition to regular load/store instructions that access memory and I/O devices mapped within the 4GB address space, MB also supports AXI4-S. AXI4-S is used with put/get instructions; its interface consists of one input and one output port, providing a low latency dedicated link to the processor's pipeline. We use AXI4-S for inter-core and core to VP connections. We use blocking and non-blocking put/get instructions. Blocking stalls MB if the receiver/sender is not ready. With non-blocking, MB keeps executing instructions even without needing acknowledgment.

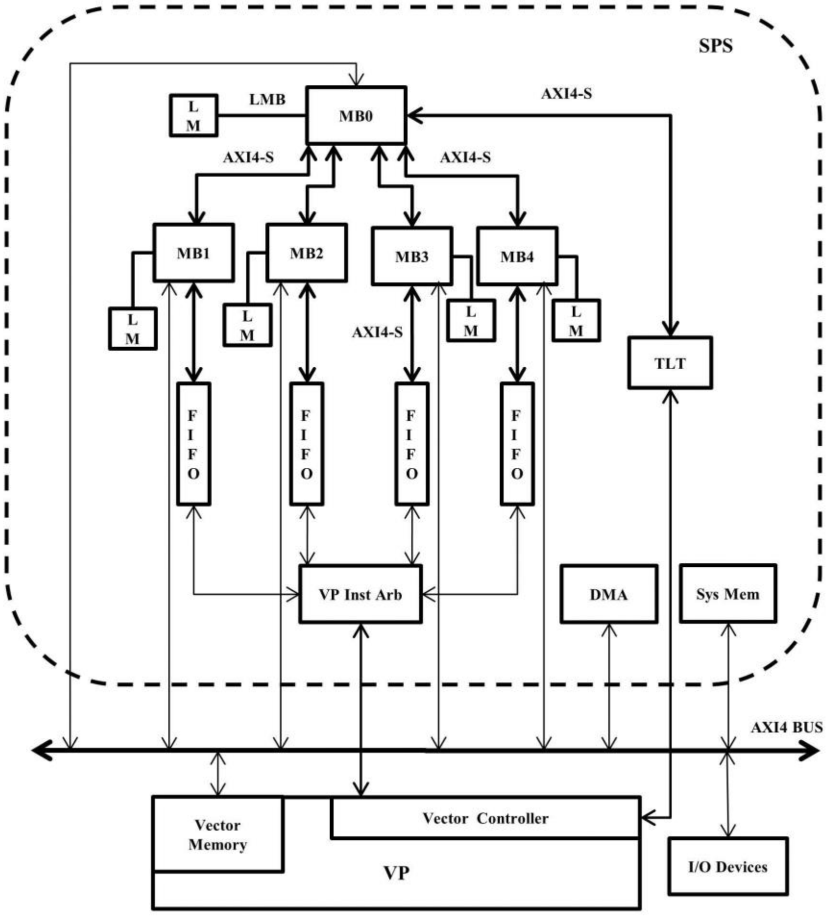

Fig. 1. Multicore architecture for VP sharing (Instr Arb: vector instruction arbitrator).

MB0 is connected to four MBs and TLT using AXI4-S. MB0 performs these tasks: i) It runs the register management algorithm for VRF virtualization. ii) It updates TLT based on the former algorithm's mapping of a thread's virtual vector registers to physical VRF registers. iii) It estimates VP utilization using information for active vector threads before scheduling new threads. iv) To simplify benchmarking on our prototype, MB0 notifies application cores MB1-MB4 about new tasks assigned to them. v) And, it polls MB1-MB4 for task completion before releasing VP resources.

MB0 is connected to TLT using the output port of its AXI4-S, and uses a non-blocking put since it knows if TLT is ready. Connections between MB0 and slave cores are bi-directional. MB0 assigns tasks to idle cores. With a non-blocking get, MB0 polls each slave for task completion, which is denoted by a task completion flag, for avoiding premature release of VP resources. MB0 is attached to a fast 32KB LM that contains the register management and thread scheduler codes.

MB1-MB4 serve as application cores (ACs) running applications that may contain function calls to vector kernels. These vector kernels are stored in a library in the attached 16KB LM. For benchmarking simplicity, ACs receive commands from MB0 to execute vector kernels and acknowledge to MB0 their successful completion. ACs apply blocking put/get to communicate with MB0. Another AXI4-S interface connects an AC to its dedicated vector instruction FIFO (see Fig. 1). An AC generating vector instructions (covered in Section 5) forwards them to this FIFO. Each vector instruction goes through the VP instruction arbitrator before reaching the VP.

Each First Word Fall Through (FWFT) vector instruction FIFO contains 16 32-bit words. An AC sends vector instructions or relevant data using blocking puts. An AC keeps issuing vector instructions until its FIFO is full. The latter condition implies VP saturation due to a round-robin arbitrator that gives equitable access to all ACs; it polls non-empty FIFOs. The pipelined arbitrator has two stages for arbitration and handshaking with the VP, respectively. FIFO and arbitrator interconnects accommodate 32-bit transfers per clock cycle.

An AXI4 connects all MBs to the vector and 128 KB system memories with separate read and write channels, and supports incremental bursts for up to 256 32-bit transfers. The vector memory is also accessible by VP. Vector data initially stored in the system memory are moved to VM for processing. A DMA engine expedites transfers. Each VM bank has two ports; one port directly connects to a lane's LDST unit. With four direct connections between VP lanes and VM banks, a four-fold bandwidth increase is achieved between VP and VM compared to a system with a crossbar [Beldianu and Ziavras 2013]. The other port of each bank is connected to the system bus in low-order interleaved fashion; sequential data communicated by a MB or the DMA engine are low-order interleaved among the four banks to support fast pipelined access. I/O devices on the system bus support debugging, display and I/O.

4. VP VIRTUALIZATION

Without loss of generality, our prototype supports simultaneous VP sharing for four threads, where a thread's VL is from the set {16, 32 and 64}. Our VP virtualization resolves register conflicts among active threads. Each vector thread is programmed with its own virtual register name space that is mapped at runtime to physical VRF registers based on their availability. Our virtualization involves two components: (1) a register management algorithm run by MB0 that determines virtual to physical vector register mappings; and (2) a hardwired TLT that facilitates the fast translation of IDs between virtual and physical registers after the former algorithm completes a mapping. With the programming interface for our prototype, applications have access to virtual vector registers 0-31 for VL=16 or 32, and 0-15 for VL=64.

The physical VRF consists of 16 vector registers where each register can store 64 (i.e., VL=64) 32-bit elements. If needed, each register of VL=64 can be split into two registers of VL=32, and each register of VL=32 can be further split into two registers of VL=16. We use the notation **reg_64(n-1)** to represent the n-th physical vector register for VL=64, where n=1, 2, …, 16. As illustrated in Fig. 2, **reg_64(0)** can be split into **reg_32(0)** and **reg_32(1)**, or further to become **reg_16(0)**, **reg_16(1)**, **reg_16(2)** and **reg_16(3)**. The vector instruction decoder needs both a register's physical name and an instruction's VL to physically locate a register in VRF. In our VP prototype, each instruction contains a 2-bit thread ID, the 5-bit IDs of involved virtual registers, and the instruction's VL encoded in a 2-bit field. The thread ID and virtual register IDs are used to obtain physical register IDs from TLT.

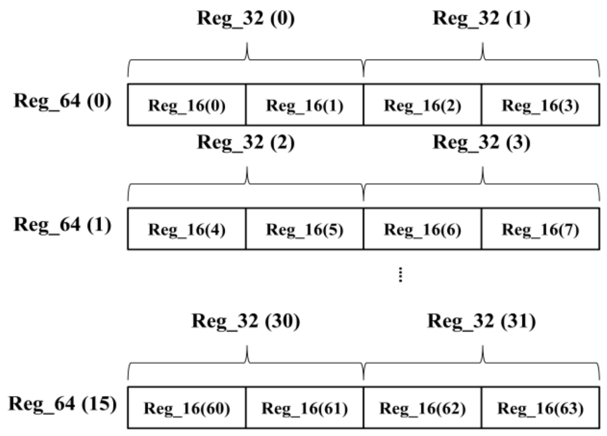

Fig. 2. VRF structure.

4.1 The Vector Register Management Module (RMM) and Algorithm

The functional blocks of the register management module (RMM) and its TLT interface are shown in Fig. 3. The register management algorithm supports a virtual space of 32 vector registers for each thread. RMM receives as input a request to either allocate or release a number of registers of certain VL; for a release, it just receives the ID of a retiring vector thread since RMM maintains detailed lists of assigned resources. After processing the request and updating TLT, RMM assigns a vector thread ID to the new allocation and sends it to the requesting core. To minimize vector register fragmentation, the register access queues as well as the register split, allocation, release and merge/recovery mechanisms give priority to the preservation of registers with larger VL. For our current benchmarking, the functionality of RMM is realized in software by MB0. A hardwired RMM is a future objective towards even higher performance and lower energy consumption. The register management algorithm is written in C.

Fig. 4 shows two data structures for VRF management. *Struct* **vp_control** contains data needed to manage VRF. Each register is an instance of *struct* **vp_reg**; there are three **vp_reg** arrays in **vp_control** for VL=16, 32 and 64, respectively. A register's **vp_reg** record is located by using its physical ID as the index into one of the three arrays. If the register is available for access, **vp_reg** can also be accessed using the quick access queue. Inside **vp_reg**, field **rname** is the physical name of the register; it initializes to the index within the array. Field **in_queue** is set to '1' when a register is put into the fast access queue; it is available to be assigned to a thread or to be split for smaller VL. After a register is assigned or split, **in_queue** is set to '0' and **used** is set to '1'. Fields **prev** and **next** are used to implement the fast access queue (a doubly linked list).

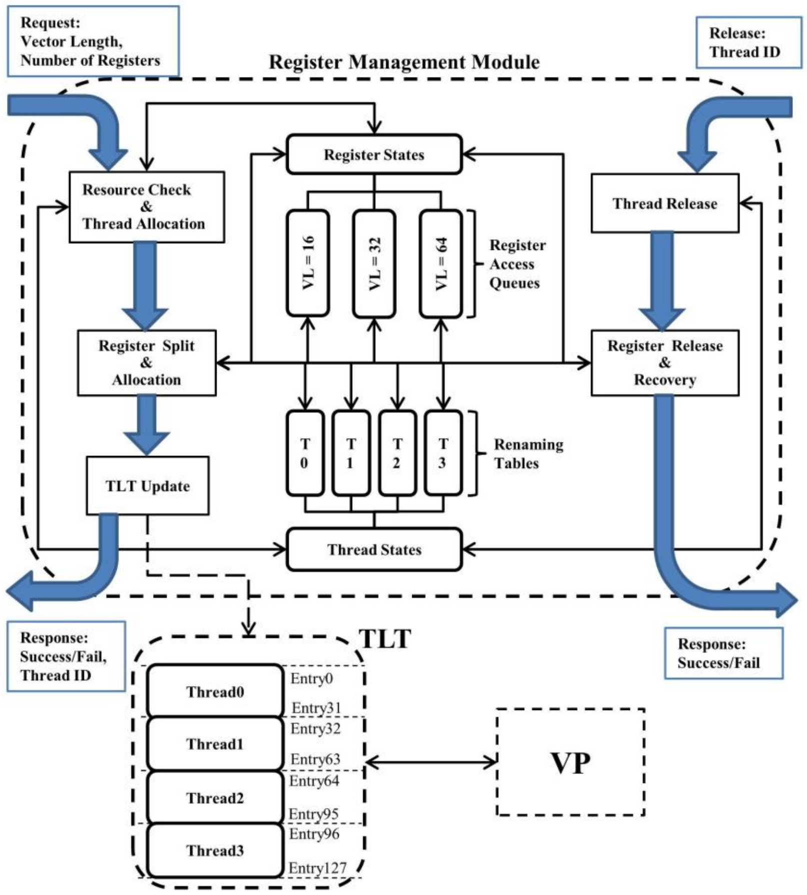

Fig. 3. RMM and its TLT interface.

The fast access queue is accessed to identify an available register for allocation or splitting. Using one of the **head_16**, **head_32** and **head_64** pointers in **vp_control**, the **vp_reg** record of the first available register in a queue is found and its fields are modified accordingly. Before any thread accesses VP, **vp_control** is initialized. No register is used initially, therefore the fields representing the number of registers available for VL=16, 32 or 64 are 64, 32 and 16, respectively. Initially, all 16 registers with VL=64 are ready to be accessed or split; therefore, they are arranged into the fast access queue pointed to by **head_64**. The other two access queues for VL=32 and 16 are initially empty. Fields **in_que_64**, **in_que_32** and **in_que_16** are initialized to 16, 0 and 0, respectively.

4.2 Assigning/Releasing VRF Resources

When a thread requests VP access, its VL and needed number of registers are provided. Based on VL's value, **avail_16**, **avail_32** or **avail_64** within **vp_control** is compared with the latter number. If the remaining number of available registers is not enough for the thread, VP access is denied. Otherwise, the thread is assigned an ID (0 to 3) for unique identification while using the VP, and register allocation begins. **thread_len[ID]** and **thread_num[ID]** in **vp_control** are modified to record the thread's VL and number of registers.

```
struct vp_reg
{
 int rname; //Register's physical name
 int in_que, used; //Register's status
 vp_reg *prev, *next; //Pointers for implementing the access queue
};

struct vp_control
{
 vp_reg reg_16[64], reg_32[32], reg_64[16]; //Array of all the registers
 vp_reg *head_16, *head_32, *head_64; //Head of access queue for each VL
 int avail_16, avail_32, avail_64; //Number of registers available for each VL
 int in_que_16, in_que_32, in_que_64; //Number of registers in the fast access queue
 int thread_len[4]; //VL for each thread
 int thread_num[4]; //Number of registers used by each thread
 int tlt_table[32][4]; //Mapping of virtual name to physical name
};
```

Fig. 4. Data structures used to manage the VRF.

Only vector registers in the fast access queue are allocated. When registers of VL=16 are needed, their available number in the queue is checked; if the number is not sufficient, registers in the queue of VL=32 are split. If registers in the queue of VL=32 are not sufficient, registers in the queue of VL=64 are split. Whenever a register of VL=N is split, for N=64 or 32, the respective number of VL=N registers in the queue and the potentially available number of registers are decremented by one. However, for registers of VL=N/2, their number in the queue is incremented by two while their number of potentially available remain unchanged until the register is actually allocated.

After register splitting, there will be sufficient registers in the fast access queue representing the VL of the assigned thread. Chosen registers are removed from the queue for allocation. The physical IDs of the registers are stored into TLT and **tlt_table** in **vp_control**. The physical names in **tlt_table** will later be used to release VP registers. TLT has three read ports and contains the same information with array **tlt_table**; it supports three VP register name readings per clock cycle. VP uses the 2-bit thread ID concatenated with the 5-bit register ID to form an index into the 128-entry TLT for locating the physical register ID used by a vector instruction.

When a thread finishes execution, the **tlt_table** entries assigned to the thread are identified for releasing its registers. Instead putting it back into the fast access queue, a released register may be combined with its “sister” register to form a register of higher VL depending on the current status of VRF. For example, **reg_16(15)** is checked when **reg_16(14)** is released. If **reg_16(15)** is not in the access queue, **reg_16(14)** is returned to the queue. Otherwise, the two registers are combined into **reg_32(7)**; it may trigger the recovery of **reg_64(4)** based on the status of **reg_32(6)**.

5. VP ARCHITECTURE

VP consists of a VC, data hazard detection unit (HDU), VRF of 1024 32-bit elements, 64KB VM, and four vector lanes; each lane has a LDST unit and a FPU. VM is divided into four low-order interleaved banks; each bank is a true dual-port RAM with one port connected to a distinct vector lane and the other port to the system bus. Each vector lane can only access its own dedicated VM bank; all cores and the DMA controller can access all four VM banks. Application data are initially stored in the system memory, and are transferred for VP processing to VM using either the DMA engine or an AC. Fig. 5 shows the architecture of our prototype. Two types of vector instructions are used. The first type is for vector-vector ALU operations and the instruction is 32 bits; it does not contain data. The second type contains a 32-bit operand in addition to the 32-bit instruction; e.g., vector-scalar ALU instructions and vector LDST instructions are of this type. Vector instructions are generated by ACs using macro definitions in C and are sent to the VP via the arbitrator interface.

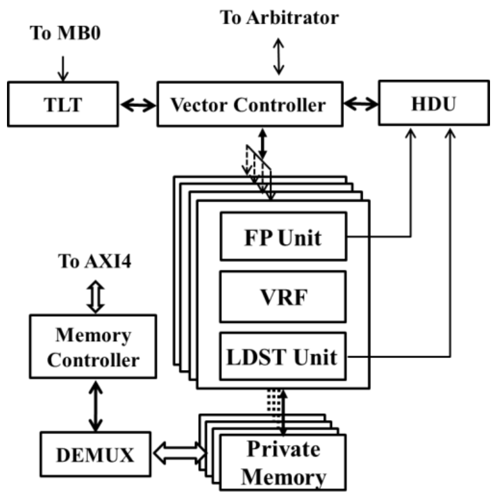

Fig. 5. Detailed architecture of the four-lane VP. (FP: Floating-point.)

5.1 Pipelined ALU and LDST Units

The first three pipeline stages in our VP's data path are inside VC, which handles register renaming, hazard detection, and assignment of ALU and LDST instructions into separate data paths. The ALU and LDST pipeline stages are shown in Fig. 6. Two clock cycles are consumed in the ALU or LDST FIFO to pass an instruction and its data to VP. The ALU decode unit consumes four clock cycles for decoding, fetching operands and feeding them to the execution unit. The FPU takes six clock cycles and an extra cycle is needed by WB. The total latency to fill up the pipeline with ALU instructions is 16 clock cycles (considering both the lane and VC delays).

Memory access instructions are decoded by the LDST decode unit, which uses six stages with store instructions for data fetching and address generation. For a load from VM, two more clock cycles are added for memory access and WB data latching. One clock cycle delay is present between the fetching of consecutive vector instructions from a FIFO. This delay represents a high impedance state ('Z'); it eases functional verification and instruction tracking in behavioral simulation. To fill up the pipeline, 11 and 13 clock cycles are needed for a store and a load, respectively. ALU and LDST instructions share the first three stages in VC.

5.2 VP-MB Interface

VC does not give VP access to the arbitrator if the lane FIFO is full or a previous instruction is stalled due to a data dependency. Register renaming is performed by reading physical register names/IDs from TLT. Each vector instruction uses at most three vector registers, and therefore TLT is triple-ported. In the renaming stage of VC, virtual names are replaced by physical names which are determined as per the VRF virtualization technique introduced in Section 4.

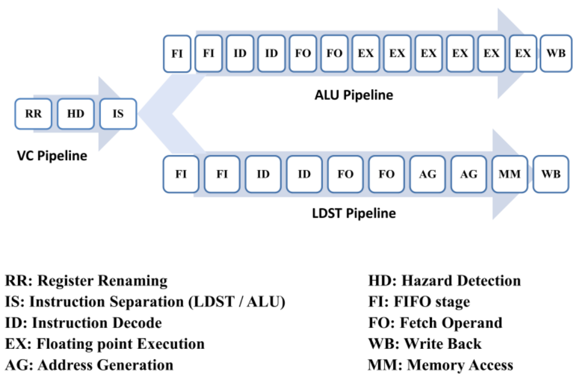

Fig. 6. Pipeline structure in the LDST and ALU data paths

5.3 Hazard Detection Unit (HDU)

After updating the register name fields, instructions enter HDU. RAW (Read-After-Write), WAW (Write-After-Write) and WAR (Write-After-Read) data hazards are detected HUD. Hazard information is forwarded to VC that may stall instructions. We assume no dependency across threads. Each HDU module has two separate slots that buffer the previous ALU and LDST instructions of a thread that entered the vector lanes, and two counters that count the number of remaining same-thread ALU and LDST instructions in the lanes. Each buffered instruction is a potential cause of hazard since an incoming instruction may depend on it. The counter of the corresponding instruction type is incremented by one upon issuing a new instruction from the same thread; it is decreased by one when an instruction of its corresponding type completes execution. The ALU and LDST units broadcast an acknowledgment with the thread ID to HDU modules when an instruction completes; the module with the matching thread ID then updates its counters. A zero count implies no pending instruction of its corresponding type in the lane for this thread; thus, there is no need to check the buffered instruction for hazards. When an instruction enters HDU, the HDU module that corresponds to the instruction's thread-ID performs hazard detection. The instruction is compared against both buffered instructions in the module; upon a data hazard detection, the instruction is stalled from entering the lanes until any related counter is reduced to zero.

5.4 Vector Lane Structure

The lane architecture is depicted in Fig. 7. To reduce the complexity in order to track the progress of instructions through pipelines, simple execution units are chosen. Once a vector instruction passes hazard checking, it is broadcasted to all vector lanes. A lane's VRF consists of 256 32-bit (single-precision FP) elements. It is accessed using three read and two write ports since the ALU and load units need two and one read port, and the WB and store units require one write port each. The design has one clock cycle latency to send an output. All read ports are configured with an "enable" for power efficiency. The ALU decode unit requires two read ports when reading a pair of operands for vector-vector operations. A lane's ALU execution unit contains a floating-point adder/subtractor and a multiplier derived from open source code. It has six pipeline stages for addition and subtraction, and four for multiplication. The results are sent to the WB block, which is connected to a write port of VRF for writing one element per clock cycle in a pipelined fashion.

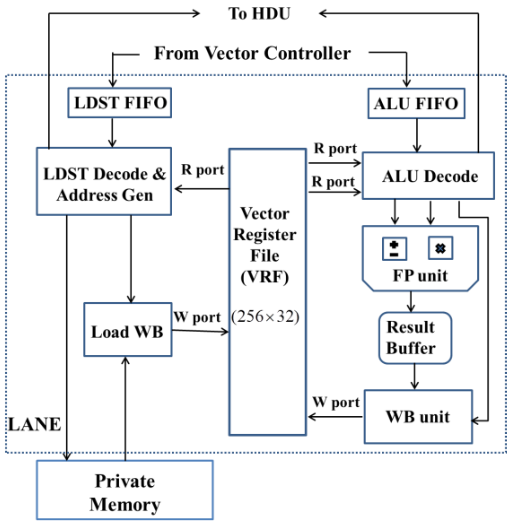

Fig. 7. Vector lane architecture.

LDST instructions use absolute memory addressing with a unit or non-unit stride. Each lane is connected to a private VM bank, therefore memory accesses are never stalled. Arbitration deteriorates performance if all memory banks are accessible to all lanes [Beldianu and Ziavras 2013]. The ALU and LDST decode blocks in each lane include counters for synchronization across lanes; counts are initialized based on the VL value in instructions. Vector instructions with different VLs may coexist in VP.

6. FPGA IMPLEMENTATION

Our prototype uses a Xilinx Virtex6 xc6vlx240t FPGA device. The entire VP, arbitrator and TLT are custom designed in VHDL. The rest of the system uses IP cores in Xilinx ISE. The system is fully synthesized and routed. The chosen clock frequency of 100MHz is the result of the open source FPU codes. Critical path delay analysis shows that the VP's clock cycle could be as low as 7.01 ns (i.e., representing 142.65 MHz) corresponding to the adder. This delay is due to 32 levels of logic. The earliest and latest signal arrival times are 1.897 ns and 2.126 ns, respectively.

Table I shows the resource consumption. This FPGA contains 37,680 slices; each slice has eight registers and four 6-input lookup tables (LUTs). Each register is implemented with flip-flops or latches, and each LUT may be composed of a pair of 5-input LUTs. Some LUTs are implemented as small RAM blocks which are known as distributed RAMs. Large RAM memory can be realized using 36Kbit BRAM blocks (RAMB36E1). Embedded DSP slices (DSP48E1) contain a hardwired 25x18 two's complement multiplier/accumulator. Our FPUs are designed with custom ASIC logic without DSP slices. Only four DSP48E1s are used, one for each lane's address calculator in the LDST unit. The VP subsystem and its SPS interface (including the vector instruction FIFOs, arbitrator and TLT) consume 13.9% and 45.8% of the total registers and LUTs. The resource consumption of FPGA-based designs is also affected by the randomness of the routing process. Some registers and LUTs are used as wires and buffers to reduce critical path delays. Our benchmarking relies on cycle accurate behavioral system simulation. For highly accurate power measurements, post place-and-route simulation is performed at a fine detail, down to the switching of individual LUTs. The binaries for each benchmark are generated and used as testbenches to obtain Switching Activity Interchange Format (SAIF) files, which are then used by the Xpower Analyzer to derive accurate power consumption.

Table I. Resource consumption.

Entity	Slice Registers (% Utilization)	Slice LUTs (% Utilization)	RAMB36E1s (% Utilization)	DSP48E1s (% Utilization)
A Vector Lane	10247 (3.4%)	17035 (11.3%)	0 (0%)	1 (<1%)
VM (4 Banks)	16 (<1%)	272 (<1%)	16 (3.8%)	0 (0%)
VC (Including HDU)	358 (<1%)	305 (<1%)	0 (0%)	0 (0%)
VP (VC+4 Lanes+VM)	41378 (13.7%)	68717 (45.6%)	16 (3.8%)	4 (<1%)
VP/SPS Interface	388 (<1%)	283 (<1%)	0 (0%)	0 (0%)
VP + VP/SPS Interface	**41766 (13.9%)**	**69000 (45.8%)**	**16 (3.8%)**	**4 (<1%)**
SPS	9962 (3.3%)	15268 (10.1%)	73 (17.5%)	23 (3%)

7. BENCHMARKING

As shown in Fig. 8, two basic types of vector instructions are sent to VP: without (**type V_instr_a**) and with a scalar operand (**type V_instr_b**). Macro definitions ease programming by providing an assembly-like VP programming interface. As an example, Fig. 8 shows the macro definition for the 32-bit *__ADD (vector-vector add)* type a instruction, and the *__VLD (unit-stride load)* and *__VST (unit-stride store)* type b instructions that hold an extra 32-bit scalar operand as address. The main function in Fig. 8 loads two 16-element vectors from VM and stores the summation result back into VM. To compile benchmarks written in C that also contain macros and assembly code for vector instructions, the MB GNU mb-gcc tool without optimization (i.e., option o0) is applied.

```
// Functions defining two types of vector instructions, with and w/o data
#define V_instr_a(instr)  asm volatile("put\t%0,rfsl1\t\n" ::"d"(instr))
#define V_instr_b(instr,data)  asm volatile ("cput\t%0,rfsl1\t\n" \
"put\t%1,rfsl1\t\n"  ::"d"(instr),"d"(data))

// Based on the above, define vector instructions as macros
// Constant X_SHIFT determines the location of field X within 32-bit instruction
#define __VADD(VDst,VSrc_1,VSrc_2,VL,Id)\
V_instr_a((OP_VADD<<OP_SHIFT)|(VDst<<DST_SHIFT)|(VSrc_1<<SRC1_SHIFT)|\
(VSrc_2<<SRC2_SHIFT)|(VL<<VL_SHIFT)|(Id<<THREAD_ID_SHIFT))

#define __VLD(VDst,BaseAddr,VL,Id)   V_instr_b((OP_VLD<<OP_SHIFT)|(VDst<<DST_SHIFT|\
(VL<<VL_SHIFT)|( Id<<THREAD_ID_SHIFT), BaseAddr)

#define __VST(VSrc,BaseAddr,VL,Id)   V_instr_b((OP_VST<<OP_SHIFT)|(VSrc<<SRC_SHIFT|\
(VL<<VL_SHIFT)|(Id<<THREAD_ID_SHIFT), BaseAddr)

int main(){                // For VL=16 & thread=0
__VLD(0,adr1,16,0);     // Load from location adr1 to r0
__VLD(1,adr2,16,0);     // Load from location adr2 to r1
__VADD(2,0,1,16,0);     // r2 ← r0+r1
__VST(2,adr3,16,0);     // Store r2 into location adr3
};
```

Fig. 8. Macros to define vector instructions.

The first benchmark is *matrix multiplication (MM)* for square matrices of size 16*16, 32*32 and 64*64. All elements on a row of the resulting matrix are calculated in each loop iteration to maximize the vectorization ratio (i.e., ratio of vector to scalar code). It multiplies a single element of the first matrix with all elements on a row of the second matrix to produce partial products. To calculate row i in the result, each element on row i of the first matrix is multiplied with the respective row in the second matrix and appropriate partial products are summed up. All multiplications are performed using scalar-vector multiplications, and additions are of the vector-vector type. Using an optimal approach, only two vector registers of size VL are needed. The results show that by increasing the dimensionality of the matrix and consequently VL, the time needed to generate one element in the result decreases slightly (due to a higher vectorization ratio). The second benchmark is *Finite Impulse Response (FIR) digital filter* that uses the outer product [Sung and Mitra 1987]. 16, 32 and 64 tap FIR filters are implemented with the input sequence having the same size as the filter; the resulting sequence has twice the input's length. A loop unrolling technique expands the kernel four times and increases the vectorization ratio. We use two vector registers of size VL. The third benchmark is *vector-dot product (VDP)* with VL= 16, 32 and 64. A vector-vector multiplication is followed by two vector-vector additions. Four VL-sized vector registers are used. The execution time of VDP is measured for an input of one pair of array having VL elements per thread.

The fourth benchmark is the *discrete cosine transform (DCT)* which is common in video processing. Since DCT is usually applied on fixed-sized pixel blocks, like 8*8 or 4*4, we perform one-dimensional 8-point DCT on blocks of size 8*8. 2, 4 and 8 adjacent blocks are used as input with VL=16, 32 and 64, respectively. Three vector registers of size VL are used. The last benchmark is *RGB to YIQ color space mapping (RGB2YIQ)*. It has the highest portion of vector code among all benchmarks and uses seven vector registers. We use configurations of VL=16, 32 and 64 to perform the calculation on a 1024-pixel block. Since the input size is independent of VL, higher VL leads to fewer loop iterations, and therefore shorter execution times.

7.1 Simulation Results

In this section only, we assume each time simultaneous VP runs of up to four threads from the same benchmark. The only exception is RGB2YIQ with VL=64 since it requires seven registers per thread while our VP has 16 registers of VL=64; we assume up to two threads for RGB2YIQ. 58 simulations are done for various VLs and degrees of multithreading. For clarity, the times for task request and register management are excluded from our measurements. Since the threads start execution at the same time and the SPS's VP interface involves a round-robin arbitrator, all threads finish execution at the same time. Table II to Table VI show the execution times and VP utilization of these benchmarks for various numbers of VL and active cores (i.e., threads). The execution times are for the input size described above.

The VP utilization with a single thread is very low for all benchmarks when VL=16; as more threads/cores are involved, the utilization improves substantially. As VL increases, the utilization of a thread increases up to a saturation point. As explained earlier, a high impedance state of one clock cycle between issuing successive instructions decreases the maximum utilization but eases the verification of functional behavior. Due to this effect, the nominal maximum utilization that can be achieved for VL=16, 32 and 64, is calculated as 80%, 88.88% and 94.11%, respectively. For a low VP-utilization benchmark, the total execution time of multiple threads may be almost the same as the benchmark's *native duration* (i.e., a thread's execution time with exclusive VP usage). When the total VP utilization with many simultaneous threads exceeds the VP's nominal maximum, all threads' execution are slowed down proportionally due to resource competition. When either the ALU or LDST unit saturates, the other unit's utilization may not increase further since ALU and LDST operations may depend on each other. Among the five basic benchmarks, MM, FIR and RGB2YIQ have higher ALU utilization that leads to VP saturation. VDP and DCT have higher LDST utilization that may lead to LDST saturation. Upon VP saturation, the slowdown amount is determined by the higher of the ALU and LDST utilizations. The performance of RGB2YIQ with VL=64 saturates for two cores although the ALU utilization is not close to the nominal maximum of 94%. It happens when threads produce high VP utilization and many data hazards, causing frequent VC stalls. For each benchmark, sequential C code with identical functionality and behavior was also run on a 100 MHz MB. The last column in the tables is the speedup of VP versus scalar core runs.

Table II. Matrix multiplication performance (input matrix size: VL*VL, 1 iteration per core).

VL	# of cores	LDST FLOP	ALU FLOP	Execution Time (μs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	4608	8192	241	53.11	4.78	8.49	84.97
	2	9216	16384	241	106.22	9.56	16.99	169.95
	3	13824	24576	241	159.33	14.34	25.49	254.93
	4	18432	32768	241	212.44	19.12	33.99	339.91
32	1	34816	65536	942	106.53	9.23	17.39	173.38
	2	69632	131072	942	213.06	18.47	34.78	346.76
	3	104448	196608	942	319.59	27.72	52.17	520.19
	4	139264	262144	942	426.12	36.96	69.57	693.53
64	1	270336	524288	3819	208.07	17.69	34.32	337.8
	2	530672	1048576	3819	416.14	35.39	68.64	675.69
	3	811008	1572864	4221	564.76	48.03	93.15	917.01
	4	1081344	2097152	5625	565.06	4805	93.20	917.5

Table III. FIR performance (input vector size: VL, 1 iteration per core).

VL	# of cores	LDST FLOP	ALU FLOP	Execution Time (μs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	576	1024	27	59.25	5.3	9.4	78.8
	2	1152	2048	27	118.51	10.6	18.9	157.4
	3	1728	3072	27	177.77	16	28.4	236.1
	4	2304	4096	27	237.04	21.3	37.9	314.8
32	1	2176	4096	51	122.98	10.6	20	153.07
	2	4352	8192	51	245.96	21.3	40	306.15
	3	6528	12288	51	368.94	32	60	459.23
	4	8704	16384	51	491.92	42.6	80	612.31
64	1	8448	16384	97	256	21.77	42.22	354.13
	2	16896	32768	97	512	43.54	84.44	708.26
	3	25344	48152	133	552.6	47.63	90.0	774.83
	4	33792	65536	177	561.17	47.72	92.56	776.29

7.2 Comparisons with Prior Works

To perform a fair performance comparison with prior works that focused on VP sharing for multicores, we choose a common reference point. Since VP speedups against host processors were listed in these works, we do the same for our system. Moreover, the chosen benchmark scenarios are similar (including identical VLs). Table VII shows comparisons with Beldianu and Ziavras [2013] that implemented an 8-lane shared VP for two cores using the CTS, FTS and VLS policies. FTS has the best performance among these policies. As per Section 2, FTS is similar to our VP sharing technique. Rooholamin and Ziavras [2015] used a VP with many similarities to ours. It utilized a hardware scheduler and a register renaming block to support VP sharing for two threads with identical VL. It relied on compiler optimizations to increase the instruction issue rate. Our virtualization yields better speedup than other techniques even with half the lanes.

Table IV. VDP performance (input vector size :VL, 1 iteration per core).

VL	# of cores	LDST FLOP	ALU FLOP	Execution Time (μs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	112	64	2.4	73.33	11.6	6.6	4.88
	2	224	128	2.4	146.66	23.2	13.3	9.77
	3	336	192	2.4	220	34.8	20	14.65
	4	448	256	2.4	293.33	46.4	26.6	19.54
32	1	288	160	3	149.33	24	13.33	8.1
	2	576	320	3	298.66	48	26.6	16.2
	3	864	480	3	448	72	40	24.3
	4	1152	640	3.4	527.05	84.7	47.05	28.58
64	1	704	448	3.6	320	48.8	31.1	13.05
	2	1408	896	4	576	88	56	23.5
	3	2112	1344	6	576	88	56	23.5
	4	2816	1792	8	576	88	56	23.5

Table V. DCT performance (input: VL/8 blocks of size 8*8, 1 iteration per core).

VL	# of cores	LDST FLOP	ALU FLOP	Execution Time (μs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	4224	2048	87	72.09	12.13	5.96	7.98
	2	8448	4096	87	144.18	24.27	11.92	15.97
	3	12672	6144	87	216.27	36.41	17.89	23.96
	4	16896	8192	87	23.85	48.55	23.85	31.95
32	1	8448	4096	87	144.18	24.24	11.57	19.2
	2	16896	8192	87	288.36	48.55	23.51	38.4
	3	25344	12288	87	432.55	72.82	32.25	57.65
	4	33792	16384	94	533.78	89	43.15	71.14
64	1	16896	8192	87	288.36	48.55	23.53	48.55
	2	33792	16384	109	460.33	77.5	37.57	77.50
	3	50688	24576	132	557.51	93.86	45.51	93.86
	4	67584	32768	176	557.51	93.86	45.51	93.86

Table VI. RGB2YIQ performance (input: 1024 pixels, 1 iteration per core).

VL	# of cores	LDST FLOP	ALU FLOP	Execution Time (μs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	6144	15360	244.2	88.05	6.29	15.72	358.13
	2	12288	30720	244.2	176.11	12.58	31.45	716.26
	3	18432	46080	244.2	264.17	18.87	41.74	1074.39
	4	24576	61440	244.2	352.23	25.16	62.9	1432.53
32	1	6144	15360	123.6	173.98	12.43	31.06	707.57
	2	12288	30720	123.7	347.68	24.83	62.08	1415.14
	3	18432	46080	155.8	414.06	29.57	73.49	1690.51
	4	24576	61440	204.1	421.44	30.10	75.25	1713.98
64	1	6144	15360	63.74	337.37	24.09	60.24	1372.07
	2	12288	30720	96.7	444.57	31.76	79.43	1808.8
	3	18432	46080	NA	NA	NA	NA	N/A
	4	24576	61440	NA	NA	NA	NA	N/A

Table VII. Speedup comparison with prior works.

SYSTEM \ BENCHMARK	MM	FIR	RGB2YIQ	VL
[Rooholamin and Ziavras 2015], 4 lanes,1 core	92.66	73.32	383.32	16
Our VP, 4 lanes, 4 cores	339.91	314.8	1432.53	
[Beldianu and Ziavras 2013], CTS, 8 lanes, 1 core	12.97	10.93	NA	32
[Beldianu and Ziavras 2013], FTS, 8 lanes, 2 cores	25.89	21.83	NA	
[Rooholamin and Ziavras 2015], 4 lanes,1 core	193.06	150.94	762.22	
Our VP, 4 lanes, 4 cores	693.53	612.31	1713.98	
[Rooholamin and Ziavras 2015], 4 lanes,1 core	403.50	360.12	1512.44	64
Our VP, 4 lanes, 4 cores	917.50	776.29	1808.80	

8. SCHEDULING VECTOR THREADS

We focus here on throughput-maximizing thread scheduling. We first profile each application to determine its ALU and LDST utilizations, as well as its native duration (i.e., its execution time with exclusive VP access). We evaluate combinations of simultaneously executing benchmarks (from our set of 15) for: **i)** A *closed system* with a fixed number of threads. **ii)** An *open system* with randomly arriving threads.

As observed in Section 7, when the ALU and LDST utilizations are both far below 90%, the performance is upper bounded by the speed of the ACs that issue vector instructions, and therefore multiple threads could share the VP with only negligible increase in the per-thread execution time. Due to the one clock cycle delay between consecutive instructions (Section 5.1), our VP's saturation threshold is not 100% but a number from 80% to 94% depending on the active threads' VLs. We assume a saturation threshold of 90% to design a scheduling algorithm that keeps the VP highly busy either with zero or minimum saturation.

In a closed system, all threads in a queue at a given time are scheduled. No new threads are added into the queue before all threads in the current queue have finished execution. Once a thread is picked by the scheduler, it keeps executing until the end, at which time its VP resources are released to any pending threads. Pending threads are arranged in descending order of their native duration. The ALU and LDST utilizations as well as the VRF usage of pending threads are provided to the scheduler as input. The scheduler keeps picking pending threads for execution until the VP has four threads, or no other pending thread can be accommodated due to unavailable VRF resources. The scheduler searches down the queue until a fitting thread is found which does not lead to saturation. If no such thread is found, the thread update mechanism ensures that the scheduler searches down the queue only once to find a fitting thread that results in minimum saturation. The scheduler always starts investigation with the first pending thread of the longest native duration. If the available VRF resources are sufficient, utilization saturation check is performed to see whether this thread will lead to an ALU or LDST overall utilization higher than 90%. If no saturation can occur, this thread is scheduled. Otherwise, it becomes the "potential thread" for scheduling. When another thread in the queue is found to lead to utilization saturation, it is compared against the currently potential thread. If the former thread can yield smaller ALU and LDST overall utilizations than the currently potential thread, then the former will replace the latter as the potential thread for scheduling. When the entire queue has been searched and all pending threads are either not fitting or lead to saturation, the currently potential thread is chosen for immediate scheduling.

8.1 Queues of Fixed Length

We tested our scheduler for a closed system with two queue sizes: 8 and 16 pending threads. Six successive schedules of random thread combinations were tested. Threads and their input data size were chosen with equal probability from the list of 15 benchmarks of Section 7. The average execution time per schedule is shown in Fig. 9. To identify the optimal solution for the six schedules with queue length 8, we applied exhaustive search (i.e., a C program produced the total execution time of all permutations of involved threads). Compared to the optimal case, which cannot be implemented in practice, our execution time is only 14.7% slower on the average and actually achieves optimality in one of the six schedules. For a queue length of eight, our average speedup is 2.83 compared to the case without VP sharing; when the queue length increases to 16, the average speedup increases to 3.33. As the length of the thread queue increases, the speedup approaches 4, which is ideal since it matches the maximum thread population. We chose one of the six schedules for each queue length to generate tables with detailed simulation information (Table VIII and Table IX).

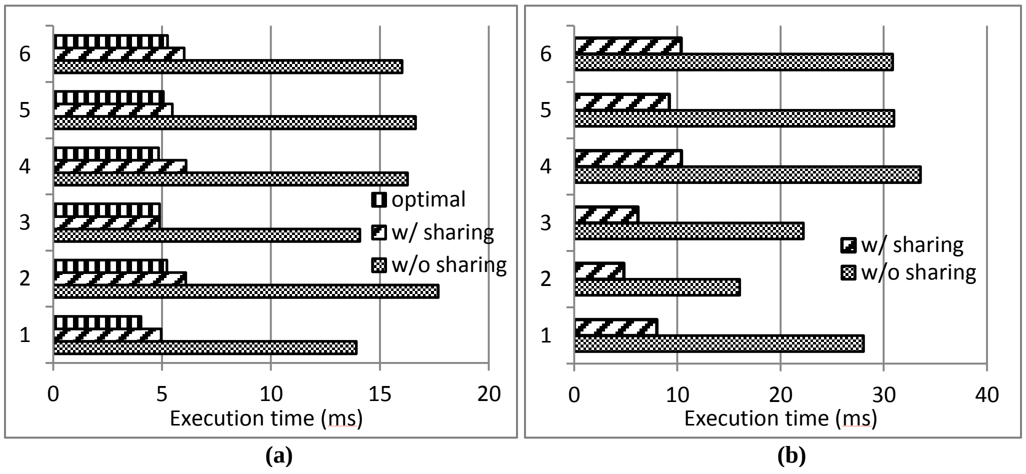

Fig. 9. Average execution time per schedule for pending thread queues of length (a) 8, (b) 16.

8.2 Open System with Randomly Arriving Threads

To simulate an open system with randomly arriving tasks, we schedule all tasks arriving within 10ms time slices. We choose a fixed input size for each benchmark to create 15 distinct tasks. The characteristics of each task are listed in Table X. Dynamic energy measurement is the focus of Section 9.1. The average task native duration is 0.182ms. Task arrival follows the Poisson distribution with a rate of λ tasks arriving per time slice. Tasks arriving in a time slice form a queue which is scheduled for execution in the next time slice. Our evaluation is for λ=0.5, 0.75 and 1; for a given λ, we generate queues for six consecutive time slices and calculate average values for the six schedules. Details of task arrivals and execution times are shown in Table XI to Table XIII. The average of the total execution time for all threads scheduled in a time slice is shown in Fig. 10. The speedup compared to the VP without sharing is 2.59, 3.15 and 3.22 for λ=0.5, 0.75 and 1, respectively. The speedups concur with the results obtained earlier for fixed thread queue lengths where the speedup increased with the thread population. Without VP sharing and scheduling, even for the lowest thread arrival rate the queue increases faster than the system can process. With our scheduling, the VP is active only 80% of the time slice for the highest λ= 1. The rest of the time the VP can be power gated to reduce the static energy (Section 9.2).

Table VIII. Detailed results for a schedule with pending thread queue length of 8.

Task ID	Application	VL	Native Duration (μs)	% ALU Utilization	% LDST Utilization	Issue Time (μs)	Commit Time (μs)	Actual Duration (μs)
0	MM	16	4820	9	5	11	4905	4894
1	VDP	64	3600	31	49	30	4348	4318
2	DCT	64	2610	24	49	3075	6083	3008
3	FIR	16	2025	9	5	44	2109	2065
4	MM	32	1884	17	9	60	1967	1907
5	RGB2YIQ	64	1268	60	24	2680	4655	1975
6	VDP	16	960	7	12	1994	3048	1054
7	FIR	32	510	20	11	2132	2642	510
Practical issue order based on static scheduling: 0,1,3,4,6,7,5,2. Optimal order based on simulation of all permutations: 0,3,6,4,1,2,5,7. Actual execution time = 6.083ms. Optimal execution time = 5.215ms. Total native duration w/o VP sharing = 17.677ms. Speedup = 2.91.								

Table IX. Detailed results for a schedule with pending thread queue length of 16.

Task ID	Application	VL	Native Duration (μs)	% ALU Utilization	% LDST Utilization	Issue Time (μs)	Commit Time (μs)	Actual Duration (μs)
0	MM	64	3819	34	18	11	3829	3818
1	MM	32	2826	17	9	24	2873	2849
2	RGB2YIQ	32	1483.2	31	12	54	1705	1651
3	MM	16	964	8	5	1111	2080	969
4	DCT	32	860	12	24	1740	2606	866
5	DCT	64	783	24	49	2632	3460	828
6	DCT	16	693	6	12	78	771	693
7	FIR	64	679	42	22	3533	4338	805
8	FIR	16	675	9	5	2101	2789	688
9	RGB2YIQ	64	634	60	24	4030	4815	785
10	VDP	32	630	13	24	3511	4357	846
11	FIR	32	561	20	11	2907	3468	561
12	RGB2YIQ	16	488.4	16	6	2837	3470	633
13	VDP	64	356.4	31	49	3863	4397	534
14	DCT	32	348	12	24	3559	3988	429
15	VDP	16	240	7	12	820	1070	250
Practical issue order based on static scheduling: 0,1,2,6,15,3,4,8,5,12,11,10,7,14,13,9. Actual execution time = 4.815ms. Total native duration w/o VP sharing = 16.053ms. Speedup = 3.33.								

Table X. Characteristics of chosen tasks for an open system.

Task ID	Application_VL	Native Duration (μs)	% ALU Utilization	% LDST Utilization	Vector Registers	Dynamic Energy (μJ)
0	RGB2YIQ_16	4884	16	6	7	766
1	MM_64	3819	34	18	2	792.3
2	MM_32	2826	17	9	2	404.1
3	RGB2YIQ_32	2472	31	12	7	535.8
4	FIR_64	1940	42	22	2	577.2
5	DCT_64	1740	24	49	3	417.8
6	DCT_32	1740	12	24	3	288.2
7	DCT_16	1740	6	12	3	207.8
8	MM_16	1446	8	5	2	152.34
9	RGB2YIQ_64	1268	60	24	7	354.4
10	FIR_32	1020	20	11	2	255
11	VDP_64	720	31	49	4	192.8
12	VDP_32	600	13	24	4	123.6
13	FIR_16	540	9	5	2	85.8
14	VDP_16	480	7	12	4	70.8

Table XI. Detailed task arrivals and execution time for λ=0.5.

Task ID	Application_VL	Number of Task Arrivals						Average
		Slice1	Slice2	Slice3	Slice4	Slice5	Slice6	
0	RGB2YIQ_16	1	1	1	1	0	0	.66
1	MM_64	1	0	0	0	0	2	0.5
2	MM_32	0	0	0	0	0	0	0
3	RGB2YIQ_32	2	0	0	0	0	0	.33
4	FIR_64	1	0	1	1	0	0	0.5
5	DCT_64	0	1	0	0	1	1	0.5
6	DCT_32	0	1	0	0	0	0	0.16
7	DCT_16	0	0	0	1	0	1	0.33
8	MM_16	3	2	1	0	0	1	1.16
9	RGB2YIQ_64	2	0	0	0	0	1	0.5
10	FIR_32	0	0	0	1	0	0	0.16
11	VDP_64	1	0	1	0	1	0	0.5
12	VDP_32	2	1	1	1	2	0	1.16
13	FIR_16	0	1	2	2	1	2	1.33
14	VDP_16	2	0	1	0	0	0	0.5
Total Native Duration (ms)		25.3	12.39	11.15	11.26	4.2	14.9	13.21
Actual Duration (ms)		8.22	4.9	4.9	4.9	1.8	4.7	4.9
Speedup		3.08	2.52	2.26	2.28	2.26	3.12	2.59

Table XII. Detailed task arrivals and execution time for λ=0.75.

Task ID	Application_VL	Number of Task Arrivals						Average
		Slice1	Slice2	Slice3	Slice4	Slice5	Slice6	
0	RGB2YIQ_16	0	0	2	0	0	0	0.33
1	MM_64	0	1	0	2	0	0	0.5
2	MM_32	2	0	0	0	1	0	0.5
3	RGB2YIQ_32	1	2	0	1	0	1	0.83
4	FIR_64	1	1	1	0	1	1	0.83
5	DCT_64	1	1	1	2	0	1	1
6	DCT_32	0	0	0	0	0	1	0.16
7	DCT_16	1	2	1	1	0	0	0.83
8	MM_16	2	3	1	1	0	2	1.5
9	RGB2YIQ_64	1	0	2	1	1	0	0.83
10	FIR_32	1	0	1	0	1	0	0.5
11	VDP_64	3	2	0	0	1	1	1.16
12	VDP_32	0	2	1	0	0	0	0.5
13	FIR_16	1	0	1	4	1	0	1.16
14	VDP_16	0	1	0	0	1	0	0.33
Total Native Duration (ms)		21.4	23.38	21.33	20.2	8.79	11.5	17.77
Actual Duration (ms)		6.59	6.75	6.66	6.62	3.05	3.75	5.57
Speedup		3.25	3.46	3.20	3.05	2.88	3.06	3.15

9. VP ENERGY CONSUMPTION

We investigate the energy consumption for the benchmarking of Section 7. Based on the power dissipation of individual benchmarks, a projection is made of the total energy consumption for the dynamic schedules of Subsection 8.2. Power consumption has three components: device static, design static and design dynamic [Beldianu and Ziavras 2014]. The device static power, also known as leakage power, is a device specific constant not related to resource utilization or switching activity. Under our simulation conditions for an ambient temperature of 50^{0}C and an airflow of 250LFM (linear feet per minute), the leakage power for our chosen FPGA is 2.88W. The design static power represents the power consumption when the device is configured but there is no switching activity. It includes the static power in I/O DCI terminations, clock managers, etc., and is related to FPGA resource consumption. The design dynamic power results from the switching of the user configured logic. Accounting for the FPGA resources that our VP actually uses, our power model adds the design's static and dynamic powers to estimate the total dissipation.

Table XIII. Detailed task arrivals and execution time for λ=1.

Task ID	Application_VL	Number of Task Arrivals						Average
		Slice1	Slice2	Slice3	Slice4	Slice5	Slice6	
0	RGB2YIQ_16	2	1	1	0	0	1	0.83
1	MM_64	1	2	2	2	2	0	1.5
2	MM_32	1	0	0	1	0	1	0.5
3	RGB2YIQ_32	0	2	3	1	1	0	1.16
4	FIR_64	1	2	0	0	0	0	0.5
5	DCT_64	2	0	1	0	1	0	0.66
6	DCT_32	1	0	1	0	2	0	0.66
7	DCT_16	0	2	2	0	2	1	1.16
8	MM_16	2	3	3	1	0	0	1.5
9	RGB2YIQ_64	1	1	0	1	1	2	1
10	FIR_32	1	0	1	1	3	1	1.16
11	VDP_64	0	2	1	0	1	0	0.66
12	VDP_32	0	2	1	1	1	2	1.16
13	FIR_16	0	1	2	1	2	2	1.33
14	VDP_16	0	2	1	0	0	1	0.66
Total Native Duration (ms)		28.75	34.57	35.14	17.81	25.53	15.76	26.26
Actual Duration (ms)		8.44	10.29	9.81	6.17	7.83	5.53	8.01
Speedup		3.4	3.36	3.58	2.88	3.26	2.84	3.23

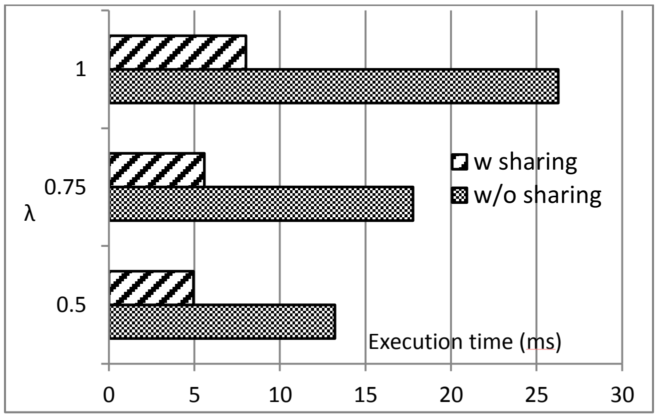

Fig. 10. The average of the total execution time for all threads scheduled in a time slice, with and without VP sharing, for λ= 0.5, 0.75 and 1. (Time slice: 10ms.)

9.1 VP Dynamic Power

To reliably estimate the dynamic power, our VP design was fully implemented and all signal switching activities of each system node were used as input for power calculation. We fully implemented (i.e., synthesized, translated, placed and routed) our VP using the Xilinx ISE tool chain, and performed post place-and-route (PAR) ISE simulations. The binaries of the vector instructions of each benchmark were generated to estimate the dynamic power. All signal switching activities during each simulation were recorded in an SAIF File. The SAIF file, with two other files generated during the implementation of the design, namely the Native Circuit Description and Physical Constraint files, were fed into the Xilinx power analyzer (XPA) to produce the VP's accurate power dissipation for each benchmark. Our power measurements include all power consumed by VP subsystems (i.e., VC, HDU, vector lanes, VRF and VM). Also, register name readings from TLT contributed to the figure.

Due to the time consuming nature of PAR simulations, we measured the average power consumption for one iteration of each vector kernel. For matrix multiplication, the innermost loop that involves three vector instructions is considered as the target kernel. It is repeated VL times to produce one row of the resulting matrix. This kernel includes one load, one vector-scalar multiplication and one vector-vector addition. For FIR filtering, the target kernel for power estimation is the internal loop which is unrolled four times, slides the coefficients four times over the input sequence, and carries out multiplications and additions to produce four elements of the result. This kernel contains twelve vector instructions: four loads, four vector-scalar multiplications and four vector-vector additions. For VDP, the kernel size depends on VL. This kernel contains 11, 14 and 18 vector instructions for VL=16, 32 and 64, respectively. For VL=16, the kernel consists of five loads, two stores, three vector-vector additions and one vector-vector multiplication. For VL=32, one load, one store and one vector-vector addition are added to the former case. For VL=64, two loads and two vector-vector instructions are added to the VL=32 case. For DCT, the inner loop which calculates the output result for one output coefficient is the kernel. This kernel contains six instructions: two loads, two stores, one vector-vector multiplication and one vector-vector addition. For RGB2YIO, the chosen kernel converts the color space for VL input pixels. It contains 21 instructions: three loads, nine scalar-vector multiplications, six vector-vector additions and three stores.

For VP power measurements of individual benchmarks, the VP is used exclusively without competition. The total dynamic energy consumed by a benchmark is actually the product of its vector kernel power consumption and its native duration. The dynamic power and energy consumptions of individual benchmarks are shown in Table XIV. The energy numbers shown are based on the input data sizes of Section 7.1. Using the measured power, we can calculate the total dynamic energy consumption of each benchmark for various native durations; this approach aids the estimation of the energy consumption in dynamic environments. The dynamic energy results for the predefined tasks of Section 8.2 were included in Table X. Using a task's average number of arrivals per time slice, we can produce its average dynamic energy consumption per slice. Fig. 11 shows that the dynamic energy consumption is related almost linearly to the task arrival rate.

Table XIV. Power and energy consumption for benchmarks.

Applic-ation	VL	Kernel Duration (ns)	VC+4Lanes+Memories Dynamic Power (mW)		Kernel Dynamic Power (mW)	Application Duration (μs)	Application Dynamic Energy (μJ)
			Signal & Logic	BRAM & IO			
MM	16	365	102.04	3.32	105.36	241	25.39
	32	405	136.96	6.04	143	942	134.7
	64	555	198.68	8.8	207.48	3819	792.3
FIR	16	895	153.68	5.44	159.12	27	4.29
	32	935	239.6	10.48	250.08	51	12.75
	64	1575	284.6	13	297.6	97	28.86
VDP	16	765	136.26	11.24	147.5	2.4	0.35
	32	1235	187.4	19.04	206.44	3	0.62
	64	2275	243.28	24.92	268.2	3.6	0.96
DCT	16	525	110.16	9.32	119.48	87	10.39
	32	605	149	16.64	165.64	87	14.41
	64	775	212.28	27.92	240.2	87	20.89
RGB2YIQ	16	1465	152	5	157	244	38.3
	32	1805	209.64	8.2	217.84	123	26.79
	64	2295	267.76	13.52	281.28	63	17.72

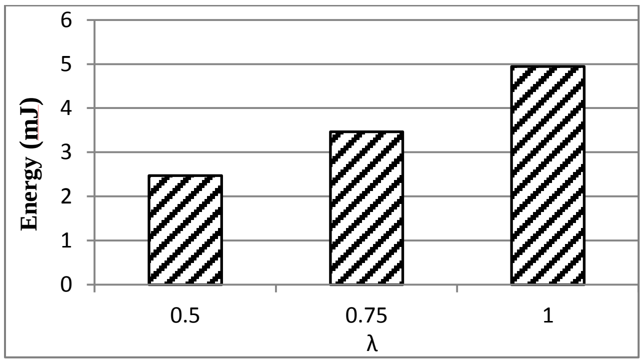

Fig. 11. Average total dynamic energy consumption per time slice for λ=0.5, 0.75 and 1.

9.2 Total Energy Consumption

The VP's static power is measured without running instructions but just applying the clock signals. For a 100µs measurement after system reset, the average static power is 214mW. Without pending instructions for the VP, power-gating (PG) can be applied to shut off the VP and zero its static power dissipation. Implementing PG requires sleep transistors, isolation cells and circuits to control power signals. It can reduce the design static power by 85% [Beldianu and Ziavras 2014].

Although commercial FPGAs currently lack PG support, PG in association with our dynamic scheduler of Section 8.2 could yield not only performance gains but also substantial reduction in the overall energy consumption. In each time slice, once the task queue becomes empty, the VP is PGed until the beginning of the next time slice. Using our static power measurements, the assumption of a 85% static power reduction with PG and the measured average execution time in Fig. 10, we project the VP's average static energy consumption per time slice for a given task arrival rate. Combining the results with the dynamic energy of Fig. 11, Fig. 12 shows the effect of PG on the VP's energy consumption with and without VP sharing. The total energy saved by combining VP sharing, proper scheduling and PG is 33.9%, 36.1% and 37% under task arrival rates of **λ=**0.5, 0.75 and 1, respectively. These are major energy savings on top of our very substantial performance improvements.

10. CONCLUSIONS

A virtualization technique was proposed for a vector coprocessor (VP) that can support simultaneous multithreading for vector processes. It can be applied in a multicore environment where the VP is shared by multiple cores via a bus or in a unicore environment where the core is designed to support simultaneous multithreading. An easy-to-use interface makes VP sharing transparent to application programmers while improving the throughput many-fold. More specifically, a VP can simultaneously execute multiple threads of similar or disparate vector lengths for improving VP throughput. We prototyped our virtualization technique for a multi-core processor embedded in an FPGA and evaluated its performance for numerous benchmarks. Under the dynamic creation of threads with diverse needs for vector sizes and types of operations, the results show impressive VP speedups of up to 333% and total energy savings of up to 37% with proper thread scheduling and power gating compared to a similar-sized system that allows VP access to just one thread at a time. Finally, our performance improvements compared to prior works for VP sharing that did not support VP virtualization further prove the viability of our approach since the obtained speedups are impressive.

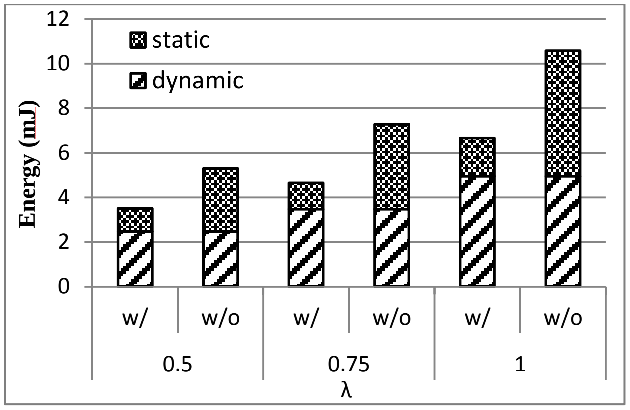

Fig. 12. Total energy consumption with (w/) and without (w/o) VP sharing, and with power gating, for λ=0.5, 0.75 and 1.

REFERENCES

Christoforos E Kozyrakis and David Patterson. 2003. Scalable, vector processors for embedded systems. *IEEE Micro. 23,* 6, 36-45.

Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. 2006. SODA: a low-power architecture for software radio. *33rd IEEE Annual International Symposium on Computer Architecture*. Boston, MA, 89-101.

Yunsup Lee Yunsup, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2013. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *ACM Transactions on Computer Systems*. *31*, 3, 6.

Junho Cho, Hoseok Chang and Wonyong Sung, (May. 2006). An FPGA based SIMD processor with a vector memory unit. *IEEE International Symposium on Circuits and Systems.* 4.

Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. (October, 2008). VESPA: portable, scalable, and flexible FPGA-based vector processors. *ACM* I*nternational Conference on Compilers, Architectures and Synthesis for Embedded Systems*. Atlanta, GA, 61-70.

Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. (2009). Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems*. 2, 1-34.

Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy Lemieux. (February, 2011). VEGAS: soft vector processor with scratchpad memory. *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 15-24.

Aaron Severance and George Lemieux.(December, 2012). "VENICE: A compact vector processor for FPGA applications." *IEEE International Conference* on *Field-Programmable Technology*. 261-268.

Aaron Severance, Joe Edwards, Hossein Omidian and Guy Lemieux. (February, 2014). Soft vector processors with streaming pipelines. *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. 117-126.

Ali R Iranpour, and Krzysztof Kuchcinski. (August, 2004). Evaluation of SIMD architecture enhancement in embedded processors for MPEG-4. *Euromicro Symposium on Digital System Design.* 262-269.

Juyup Lee, Sungkun Moon, and Wonyong Sung. (December, 2004). H. 264 decoder optimization exploiting SIMD instructions. *IEEE Asia-Pacific Conference on Circuits and Systems.* 2, 1149-1152.

Yong-Hwan Kim, Jin-Woo Yoo, Seong-Won Lee, Joonki Paik and Byeongho Choi. (January, 2005). Optimization of H. 264 encoder using adaptive mode decision and SIMD instructions. *IEEE International Conference on Consumer Electronics*. 289-290.

Yu Shengfa, Chen Zhenping and Zhuang Zhaowen. (June, 2006). Instruction-level optimization of H. 264 encoder using SIMD instructions. *IEEE International Conference on Communications, Circuits and Systems*. 126-129.

Joohyun Lee, Gwanggil Jeon, Sangjun Park, Taeyoung Jung and Jechang Jeong. (June, 2008). SIMD optimization of the H. 264/SVC decoder with efficient data structure. *IEEE International Conference on Multimedia and Expo*. 69-72.

Wing-Yee Lo, Daniel Pak-Kong Lun, Wan-Chi Siu, Wendong Wang and Jiqiang Song. (2011). Improved

SIMD architecture for high performance video processors. *IEEE Transactions on Circuits and Systems for Video Technology*. *21,* 12, 1769-1783.

Hongyan Yang and Sotirios G. Ziavras. (September, 2005). FPGA-based vector processor for algebraic equation solvers. *IEEE International System on Chip Conference.* 115-116.

Timothy Heil, Anil Krishna, Nicholas Lindberg, Farnaz Toussi and Steven Vanderwie. (2014). Architecture and performance of the hardware accelerators in IBM's PowerEN processor. *ACM Transactions on Parallel Computing*.

Spiridon F Beldianu and Sotirios G. Ziavras. (2013). Multicore-based vector coprocessor sharing for performance and energy gains. *ACM Transactions on Embedded Computing Systems*. *13,* 2.

Spiridon F Beldianu and Sotirios G. Ziavras. (March, 2014). Performance-energy optimizations for shared vector accelerators in multicores. *IEEE Transactions on Computers*. *64,* 3, 805-817.

Seyed A. Rooholamin and Sotirios G. Ziavras. (June, 2015). Modular vector processor architecture targeting at data-level parallelism, *Microprocessors and Microsystems. Elsevier,* 39, 4, 237-249.

Wonyong Sung and Sanjit K. Mitra. 1987. Implementation of digital filtering algorithms using pipelined vector processors. *Proceedings of the IEEE. 75,* 9, 1293-1303.

Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, Favid A. Koufaty, J. Allen Miller and Michael Upton. (Febr, 2002). Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*. *6,* 2, 1–12.