Microprocessors and Microsystems (Elsevier journal), accepted for publication, 2015

Modular Vector Processor Architecture Targeting at Data-Level Parallelism

Seyed A. Rooholamin and Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102, USA
Ziavras@njit.edu

Abstract:
Taking advantage of DLP (Data-Level Parallelism) is indispensable in most data streaming and multimedia applications. Several architectures have been proposed to improve both the performance and energy consumption for such applications. Superscalar and VLIW (Very Long Instruction Word) processors along with SIMD (Single-Instruction Multiple-Data) and vector processor (VP) accelerators, are among the available options for designers to accomplish their desired requirements. We present an innovative architecture for a VP which separates the path for performing data shuffle and memory-indexed accesses from the data path for executing other vector instructions that access the memory. This separation speeds up the most common memory access operations by avoiding extra delays and unnecessary stalls. In our lane-based VP design, each vector lane uses its own private memory to avoid any stalls during memory access instructions. The proposed VP, which is developed in VHDL and prototyped on an FPGA, serves as a coprocessor for one or more scalar cores. Benchmarking shows that our VP can achieve very high performance. For example, it achieves a larger than 1500-fold speedup in the color space converting benchmark compared to running the code on a scalar core. The inclusion of distributed data shuffle engines across vector lanes has a spectacular impact on the execution time, primarily for applications like FFT (Fast-Fourier Transform) that require large amounts of data shuffling. Compared to running the benchmark on a VP without the shuffle engines, the speedup is 5.92 and 7.33 for the 64-point FFT without and with compiler optimization, respectively. Compared to runs on the scalar core, the achieved speedups for this benchmark are 52.07 and 110.45 without and with compiler optimization, respectively.

Keywords: Parallelism, vector processor, performance, speedup, benchmarking.

1. Introduction

High-performance computing processors often have a superscalar or VLIW architecture that focuses mostly on exploiting ILP (Instruction-Level Parallelism). Roger et. al.[1] show that ILP and DLP can be merged in a single simultaneous vector multithreaded architecture for higher performance. VIRAM's [2] basic multi-lane architecture can be used to build VPs that exploit DLP through SIMD processing. Each lane contains similar pipelined execution and load-store units. Each vector register is uniformly distributed among the lanes. All the elements from a vector in a lane are processed sequentially in its pipelined units while corresponding elements from different lanes are processed simultaneously. Using EEMBC benchmarks, it was demonstrated that a cache-less VIRAM is much faster than a superscalar RISC or a cache-based VLIW processor [3].

The SODA VP has a fully programmable architecture for software defined radio [4]. Using SIMD parallelism and being optimized for 16-bit computations, it supports the W-CDMA and IEEE802.11a protocols. Embedded systems using a soft core or hard core processor for the main execution unit also have the option to attach a hardware accelerator to increase their performance for specialized tasks. Sometimes these accelerators are realized using FPGA (Field-Programmable Gate Array) resources to speed up applications with high computational cost. Designing a custom hardware accelerator that will yield outstanding performance needs good knowledge of HDL (Hardware Description Language) programming. Another SIMD, FPGA-based processor uses a 16-way data path and 17 memory blocks as the vector memory in order to perform data alignment and avoid bank conflicts [5]. VESPA [6] is a portable, scalable and flexible soft VP which uses the same instruction set as VIRAM but the coprocessor architecture was hand-written in Verilog with built-in parameterization. It can be scaled with regards to the number of lanes and yields x6.3 improvement with 16 lanes for EEMBC benchmarks compared to a one-lane VP. It is flexible as the size of the vector length and its width, as well as the memory crossbar, can vary according to the target application. The VIPERS soft VP is a general-purpose accelerator that can achieve a x44 speedup compared to the Nios II scalar processor [7]; it increase the area requirements 26-fold. It supports specific instructions for the applications, such as motion estimation and median filters, and can be parameterized in terms of number of lanes, maximum vector length and processor data width. VEGAS [8] is a soft VP with cache-less scratchpad memory instead of a vector register file. It achieves x1.7-3.1 improvements in the area-delay product compared to VESPA and VIPERS. With the integration of a streaming pipeline in the data path of a soft VP, a x7000 times speedup results for the N-body problem [9].

An application-specific floating-point accelerator is built using a fully automated tool chain, co-synthesis and co-optimization for SIMD extension with a parameterizable number of vector elements [10]. An application-specific VP for performing sparse matrix multiplication was presented in [11]. IBM’s PowerEN processor integrates five hardware application specific accelerators in a heterogeneous architecture to perform key functions such as compression, encryption, authentication, intrusion detection and XML processing for big workload network applications. Hardware acceleration facilitates energy-proportional performance scaling [12]. An innovative lane-based VP which can be shared among multiple cores in a multicore processor was proposed in [13]; it improves performance while maintaining low energy cost compared to a system with exclusive per-core VPs. Three shared-vector working policies were introduced for coarse-grain, fine-grain and exclusive vector-lane sharing, respectively. Benchmarking showed that these policies yield x1.2-2 speedups compared to a similar cost system where each core contains its own dedicated VP.

A major challenge with these VPs is slow memory accesses. Comprehensive explorations of MIMD, vector SIMD and vector thread architectures in handling regular and irregular DLP efficiently confirm that vector-based microarchitectures are more area and energy efficient compared to their scalar counterparts even for irregular DLP[14]. Lo et. al. [15] introduced an improved SIMD architecture targeted at video processing. It has a parallel memory structure composed of various block sizes and word lengths as well as a configurable SIMD architecture. This structure can perform random register file accesses to realize complex operations, such as shuffling, which is quite common in video coding kernel functions. A crossbar is located between the ALU (Arithmetic Logic Unit) and register file.

In a VIRAM-like architecture, a memory crossbar often connects the lanes to the memory banks to facilitate index memory addressing and data shuffling. This crossbar adds extra delay when not actually needed, such as for stride loads and stores. Moreover, it increases the energy consumption. Adding a cache to each lane may solve this problem to some extent but the cache coherence problem will require an expensive solution, often prohibitive for embedded systems. Since in practical applications stride addressing is more common than other types of addressing [16], we introduce here a VP model that does not sacrifice performance for less likely memory access instructions. We develop a VIRAM-based, floating-point VP embedded in an FPGA that connects to a scalar processor. This VP comprises four vector lanes, and provides two separate data paths for each lane to process and execute load and store operations in the LDST (Load-Store) unit in parallel with floating-point operations in the ALU. Each cache-less lane is directly attached to its own local memory. Data shuffle instructions are supported by a shuffle engine in each lane which is placed after the lane's local memory and connects to other lanes via a combinational crossbar. All the local memories connect to the shared bus which is used to exchange data between these memories and the global memory. The prototyping of a system with four lanes shows substantial increases in performance for a set of benchmarks compared to similar systems that do not contain the shuffle engines.

Previously proposed VPs are not versatile enough in multithreading environments. They were mostly capable of handling simultaneously multiple threads using the same vector length in predefined contexts. However, this approach is not often efficient for real applications since a VP is a rather high-cost, high-performance accelerator that consumes considerable area and energy in multicore processors. A more flexible VP that can be shared dynamically by multiple cores results in better resource utilization, higher performance and lower energy power dissipation. Our proposed solution supports the simultaneous processing of multiple threads having diverse vector lengths. In fact, the vector lengths used by any given thread are allowed to change during execution. The following sections show the detailed architecture of our VP, benchmarking results on an FPGA prototype, and performance analysis.

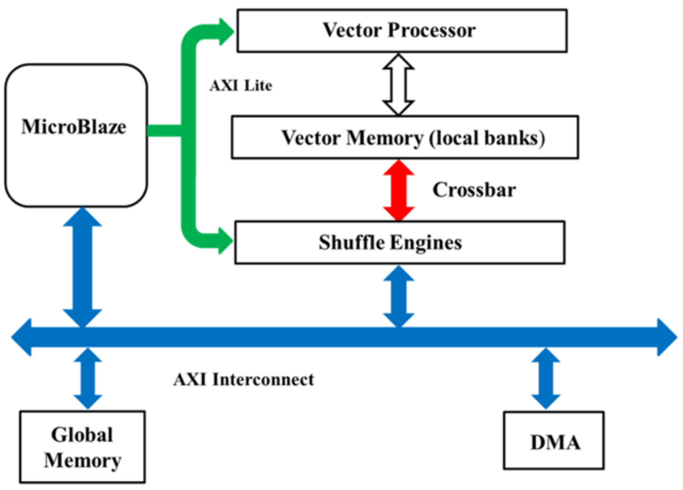


Figure 1: High-level architecture of the multi-lane VP prototyped on a Xilinx FPGA. The vector memory is low-order interleaved. Each vector lane is attached to a private memory.

2. Methodology and Realized Architecture

2.1 System Architecture Prototyping on FPGA

Figure 1 depicts the basic architecture of the FPGA-prototyped VP introduced in this paper. For the sake of simplicity we show a single scalar core, Xilinx's soft core MicroBlaze (MB) that fetches instructions from its instruction memory (not shown in the figure) and issues them to appropriate execution units. The MB is in charge of executing all scalar and control instructions while vector instructions are sent to the VP. The shuffle engine, which is distributed along the lanes, is activated only to realize vector data shuffling with multiple vector lanes. Our design introduces two innovative concepts. First, it removes the competition of lanes to access memory banks, which is the case for earlier works, by employing cache-less private memories for the lanes; the private memories form a low-order interleaved space that resides between the lanes and the global memory. Second, the vector length can vary even between instructions in the same thread. In all previously introduced VPs, the vector length was defined for each working context, program or thread. It was usually a fixed number for each thread and was set in advance by the scheduler. In contrast, our model allows us to define the vector length for each individual instruction. As a result, the vector length can vary widely, even for instructions in the same loop. Although usage of a mask register could potentially have the same effect, the performance can degrade.

Data needed by applications running on the VP should be preferably stored in the private memories of lanes. Since these private memories connect to the AXI (Advanced eXtensible Interface) shared bus, copying the data from the global memory could be done either by the MB or the DMA engine as both have access to the shared bus. If the instruction and data caches also of the MB are placed on the AXI interconnect, the time needed to copy the data from the global memory to either the vector memory or the MB data cache will basically be the same. The same principles are applied for writing back from the VP private memories or the MB data cache to the global memory. Block data are placed in consecutive locations in the MB data cache while low-order interleaving among lanes is used for the vector memory.

To evaluate the proposed VP model, we created an FPGA prototype with four lanes and four on-chip memory banks that serve as local memories. Our VP model is modular and can be easily extended to include more lanes. We used the Xilinx Virtex6 xc6vlx240t-FF748 FPGA device. To reduce the complexity of the hardware design in order to track operations progressing through the data path, we included rather simple execution units in the vector lanes. Since each lane directly connects only to its private memory in order to avoid contention when accessing memory banks, a very fast load-store unit was designed in each lane as there is no chance of stalling during memory access instructions. Contention when accessing a memory bank can only happen in the case of data shuffle instructions which, however, are totally handled by each lane's shuffle engine. Since the distributed shuffle engines employ other ports of the private memory banks than those that connect to lanes, other vector instructions can be executed while realizing data shuffling as long as no data hazard exists between the involved instructions. Figure 2 shows the detailed architecture of our prototype.

The hardware design of the vector lanes, vector controller, scheduler, data shuffle controller, data shuffle engines, and combinational crossbar and mux was done by writing VHDL code. Xilinx IPs (Intellectual Properties) were used for the realization of the memory banks, memory controllers, MB, and AXI4 and AXI4 Lite interconnects. The VP was developed using Xilinx ISE version 14.5. The MB was added to the project using the EDK tool while its configuration and connection to the peripherals was done using Xilinx XPS. The current paper focuses on proof of concept, so the prototyped VP consists of four lanes and has 1024 32-bit registers in the vector register file. It also contains a 64Kbyte vector memory that can accommodate our largest benchmark. The rest of this section contains VP details.

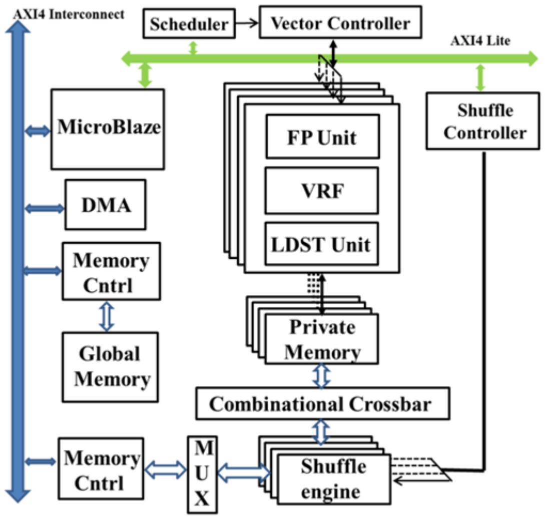


Figure 2: Detailed architecture of the four-lane VP (FP: Floating-point).

The MB soft core is a 32-bit RISC Harvard architecture that supports the AXI4 and LMB (Local Memory Bus bus interfaces. We implemented version 8.40.a with five pipeline stages and a floating-point unit. Data and instruction caches can be connected to either bus. For flexibility, we connect the memory blocks to both the AXI4 and LMB buses. Each bus requires its own memory controller. We use one AXI4 memory controller to create a slave interface for the vector memory. The AXI4 interconnect is good as a shared bus for high-performance memory mapping and can support up to 16 slaves and 16 masters [17]. The AXI4 crossbar can realize every transfer between interconnected IPs, like memories. Moreover, it supports the DMA-based burst mode for up to 256 data transfer cycles which is suitable for transfers between the global and private memories.

To connect the VP and shuffle controller to the MB for vector instruction transfers from the MB, the AXI4 Lite interconnect is used which is appropriate for this type of non-DMA memory-mapped transfers. The slave interfaces for connecting the VP and shuffle controllers to the shared bus are developed using the create-and-import peripheral wizard in Xilinx XPS. They both contain control registers which can be read and written by the MB through the AXI4 lite interconnect. A hardwired scheduler for accessing the VP is included in the VP interface. The main responsibility of the scheduler is to grant VP access to a requesting MB based on the vector length it asks for and the current availability of VP vector registers. Vector instructions are written into the VP using memory mapping.

2.2 VP Architecture and Instructions

Two types of vector instructions are used by our VP. The first type does not contain data and all the required fields for executing the instruction are placed in the 32-bit instruction; vector-vector ALU instructions are of this type. The other instruction type consumes 64 bits that contain a 32-bit operand value; e.g., vector-scalar ALU instructions are of this type. Since our main focus here is proof of concept for the hardware design, we did not develop an advanced compiler for the VP. Inline function calls are included in the C code for the MB; they represent VP instructions and their realization involves macros.

Since the VP's instruction input port is viewed as a memory location by the MB in this memory mapped system, a small delay may occur between issuing an instruction of the second type and the arrival of the needed operand. Thus, the scheduler sends them together to the VP when the data becomes available.

Every MB that has access to the AXI4 Lite interconnect can send a request to the scheduler for VP resource access. Each MB can access the VP as a peripheral device using two different addresses, for sending a VP request or release instruction and a vector instruction upon VP granting, respectively. Requests are granted by the scheduler. Threads initiate a request to the scheduler in advance using a 32-bit instruction. This request instruction includes the vector length (VL) per register and the number of vector registers needed by the thread. An affirmative reply by the scheduler will include a 2-bit thread ID that can be used to get VP access. This will occur only if there is enough space in the vector register file to accommodate the request. The MB running the thread will include this ID in all vector instructions sent to the VP. Vector register renaming and hazard detection rely on this type of ID. If the aforementioned conditions for the thread are not satisfied, the scheduler will reject the thread request with information about the currently available VL and vector registers. Although our hardware implementation allows different vector instructions to employ different VLs, a complicated register renaming unit will be needed. Therefore, for the sake of simplicity, this paper assumes that the VP can handle two threads at a time, from the same or different MB cores, where all instructions in both threads use the same VL. Otherwise, it will be the compiler's or programmer's responsibility to employ registers that will guarantee no conflicts in the VRF (Vector Register File). Threads release VP resources by issuing a release instruction to the scheduler.

The VP scheduler interfaces the VP via the VP controller (VC). The latter has a pipelined architecture that consumes three clock cycles for register renaming and hazard detection. Since the VP connects to AXI4 Lite via the shared bus, it can receive instructions from any scalar processor that connects to that bus in a multicore environment; thus, the VC unit can accept vector instructions from a multitude of threads and carry out register renaming, if needed. RAW (Read-After-Write), WAW (Write-After-Write) and WAR (Write-After-Read) data hazards are resolved by the hazard detection unit in the VC. This unit resolves all possible hazards in accessing vector registers in the lanes by using an appropriate instruction tagging mechanism. Adding a tag to each instruction allows handshaking between the VC and VP. The same instructions are issued simultaneously to all four lanes.

The detailed architecture of each lane is depicted in Figure 3. The data paths for memory and ALU instructions are completely separated in each lane, and related instructions and data are queued in different FIFOs. All the instructions and data in a lane are represented using 32 bits. Memory accessing instructions always contain 32-bit additional data to represent the private memory base address to be used. ALU instructions for vector-scalar operations also contain a 32-bit floating point scalar. ALU instructions are decoded by the ALU decode unit and the needed operands are fetched from the VRF. The VRF in each lane consists of 256 32-bit locations that can store 256 single-precision floating-point vector elements. It is accessed using three read and two write ports since the ALU and load (part of the load-store LDST) units need two and one read port in order to simultaneously read two and one operand, respectively, and the register WB (Write-Back) and store (part of LDST) units require one write port each. In the case of contention, when different ports want to perform different tasks simultaneously on the same location in the VRF, the write first policy could be applied. The design results in one clock cycle latency for sending the output to related ports; it uses output enable ports to ease the reading task. Reading from the VRF is possible only when the output enables are triggered. The ALU decode unit requires two read ports when reading a pair of floating-point operands to realize vector-vector instructions. The ALU execution unit in the lane contains a floating-point adder/subtractor and a multiplier that were developed using open source code [18]. This unit has six pipeline stages for addition and subtraction, and four stages for multiplication; it performs operations on 32-bit single-precision floating-point data. The results of the execution unit are sent to the WB block which connects to a write port of the VRF for writing one element per clock cycle in a pipelined fashion.

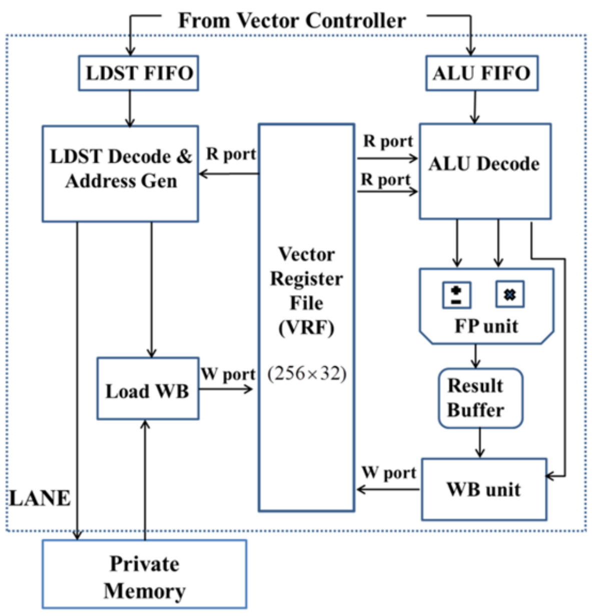


Figure 3: Lane architecture.

Absolute and indexed memory addressing are used to access the private memories. Absolute addressing may employ a non-unit stride. The LDST unit fetches the register content for a store instruction from the VRF and generates the destination address for the lane's private memory using the base address that arrived right after the instruction. It uses only one VRF read port. Each vector memory instruction issued to the lane has two 32-bit fields. The first field contains the source or destination register and the stride value, whereas the second one is a base address in the lane's private memory. Indexed addressing for the private memory is realized using the data shuffle engines. For load instructions, the WB unit writes the fetched memory contents into the proper register using a write port at the rate of one element per clock cycle. In our prototyped VP with four lanes, 1024 (i.e., 4 lanes *256 elements/lane) vector elements can reside in the VRF; the VRF is divided evenly among the four lanes so the VL must be a multiple of four. Hence, we can configure the VRF as 16 vector registers with VL=64, or 32 registers with VL=32, or 64 registers with VL=16. The location of register elements in the VRF depends on the VL value and the register ID. In the case of VL=64, for example, register "r0" contains all the elements of "r0" and "r1" for VL=32.

The ALU and LDST decode blocks in each lane include counters for synchronization when reading from the VRF and feeding the data to the next block; they are initialized based on the VL assumed by the instruction. Since our design avoids memory stalls by making a private memory available to each lane, all lanes remain synchronized in the full pipeline utilization mode where one element is processed every clock cycle in the lane. This synchronization flexibility allows dynamic changes of VL's value for any given instruction. For example, the vector-vector instruction "r2 <= r0+r1" for VL=32 can be substituted by the two vector-vector instructions “r4<=r0+r2” and “r5<=r1+r3 for VL=16, and vice versa, within a thread or a loop since the corresponding registers include the same elements from the VRF (as per the preceding paragraph).

For memory access instructions without data shuffling, the shuffle engine adds no delay since the combinational crossbar is placed in the middle of the connection between the AXI4 shared bus and the memory port. Since both the shuffle engine and the MB use the same memory ports when accessing the private memory, only one of them can write to or read from the memory in any given clock cycle; the access decision is made by the shuffle engine and is realized via the crossbar. Each lane uses independent ports to access its private memory and the LDST unit can execute the next memory access instruction while data shuffling is performed as long as there are no data hazards. If there is an access contention on a memory bank while running a data shuffle instruction, the shuffle engine will apply the round robin scheduling policy. Indexed memory addressing also can be realized by the shuffle engine. The shuffle controller simultaneously provides to all four shuffle engines the information needed for shuffling (i.e., the source, destination and index register values).

2.3 Pipelined ALU and LDST Units

The VC as well as the vector lanes are pipelined. The first block in our VP’s data path is the VC which has three pipeline stages for register renaming, hazard detection, and separating for forwarding the ALU and LDST instruction word components (e.g., base address or scalar operand), respectively. Two clock cycles are consumed in either FIFO to pass an instruction and its data to the VP. The ALU decode unit consumes four clock cycles for decoding, fetching operands and feeding them to the execution unit. The floating-point execution unit consumes six clock cycles for processing and an additional cycle to receive an acknowledgment from the WB unit after writing a result into the VRF. Thus, the total latency for filling up the pipeline with ALU instructions is sixteen clock cycles (accounting for all delays in the lane and VC), as shown in Figure 4 (the first three stages are inside the VC).

Memory access instructions are decoded by the LDST decode unit which contains six pipeline stages for instruction decoding, data fetching from the VRF and address generation when executing store instructions. For a load involving the private memory, two more clock cycles are added representing a memory access and data latching by the WB unit, respectively. There is also one clock cycle delay between fetching consecutive vector instructions from either FIFO. This delay eases functional verification and instruction tracking through the data path during behavioral simulation, since it represents a high-impedance state (‘Z’) delineating consecutive instructions. The total latency for filling up the pipeline is 11 and 13 clock cycles for a store and a load instruction, respectively, as shown in Figure 4. For data shuffle instructions, the data path consists of the shuffle controller and shuffle engines. For a shuffle instruction, the shuffle controller accepts three addresses representing the location of the source, destination and index data in the vector memory. This controller will not initiate data transfers until all the required information for the desired permutation becomes available. After sending the information to the shuffle engines, four clock cycles are needed per element to fetch the data and the corresponding index from the memory, and at most four more clock cycles to apply round-robin scheduling upon data collision involving any of the four private memories.

The FPGA-based simulation testbench was built using the Xilinx Project Navigator. The chosen working frequency of 50 MHz for the VP is the result of the open source codes used to implement the ALU’s floating-point execution unit. However, critical path delay analysis shows that the VP’s clock cycle could become as low as 7.01 ns (i.e., a frequency of 142.65 MHz) corresponding to the path delay in 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. A 50 MHz frequency was thus chosen for the MB and all the peripherals (e.g., memories, memory controllers and VP).

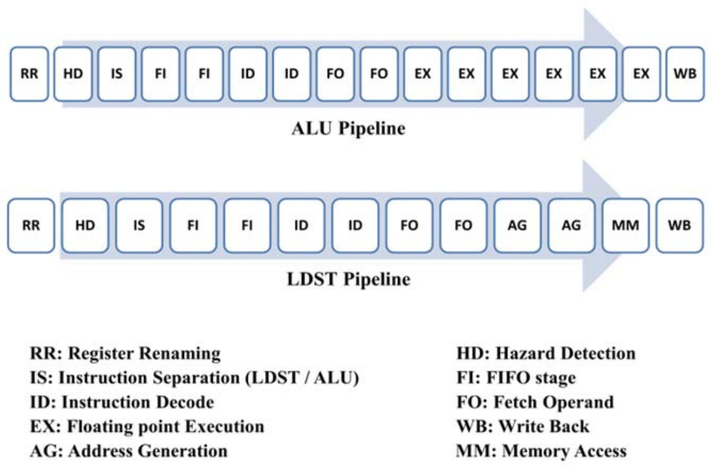


Figure 4: Pipelined structures in the ALU and LDST data paths.

2.4 Resource Utilization

Before demonstrating the proposed architecture's performance achievements, it is essential to know the silicon area occupied by this design. We synthesized the architecture of Figure 2 for the Virtex6 xc6vlx240t-FF748 FPGA device which is organized in columns and is built with a 40nm copper CMOS process. This Xilinx device includes 37,680 slices, where each slice contains 4 LUTs and 8 flip flops for realizing configurable logic. It also includes 768 DSP48E1 DSP modules, where each module contains an 25*18-bit multiplier, an adder and an accumulator. There are also 344 block RAMs (BRAMs) of 36 Kbits each which are used to realize memory components in digital designs. The overall resource consumption of our design is presented in Table 1. The MB system consumption in the table is without the VP and the connection interfaces of Figure 2. We can conclude that the VP accelerator consumes almost 11times as much area as the MB in an effort to speed up data-parallel applications. Also, the data shuffle engines do not consume many resources. We will extrapolate further in a subsequent section by investigating the dynamic energy consumption of these resources for a set of benchmarks.

Table 1: Resource consumption.

Entity	Slice Registers (% Utilization)	Slice LUTs (% Utilization)	RAMB36E1s (% Utilization)	DSP84E1s (% Utilization)
Vector Processor	45212 (14.9%)	69127 (45.8%)	0	4 (0.5%)
Vector Memory	2 (0%)	296 (0%)	16 (3.8%)	0
Shuffle Engines	1320 (0.4%)	1228 (0.8%)	0	0
MB System	4947 (1.6%)	6183 (4.1%)	16 (3.8%)	3 (0.4%)

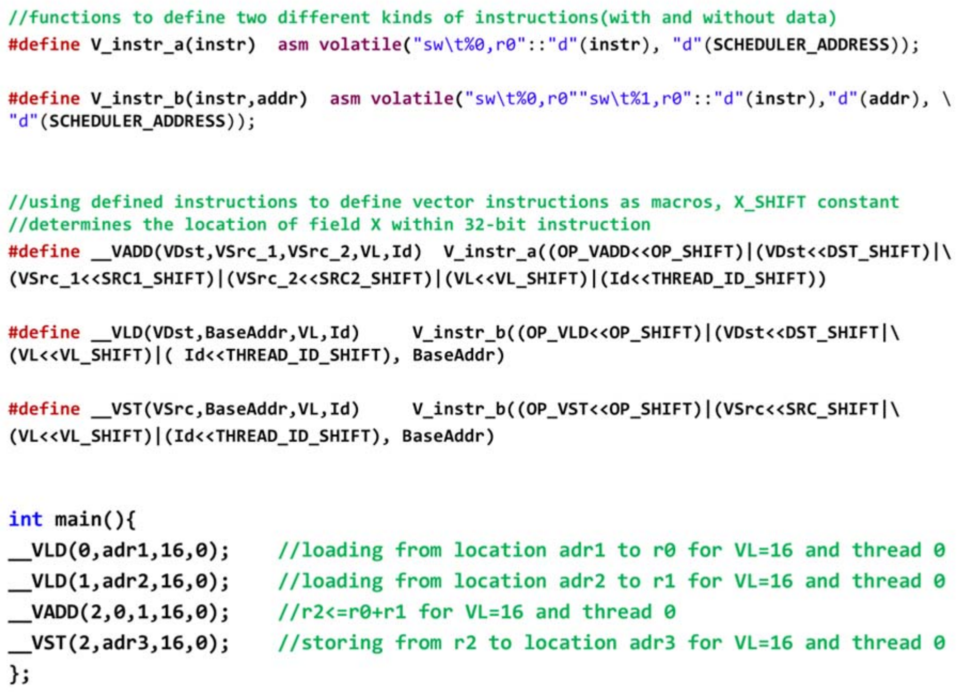
3. Performance Benchmarking

We employed various vector-intensive benchmarks to evaluate our design. Since MB is a soft core processor, the simulation of the executable file is performed using the developed RTL model. By performing behavioral simulation, we can get access to all the ports, signals and memories in the system. We integrate all the system components using the ISE project navigator and all the connections are made according to the architecture described in the previous section. The designed hardware is exported to the SDK tool so that the execution of developed application benchmarks can be driven by the scalar processor. The inline embedded macros for the VP are hand-coded to maximize its performance. Also, the drivers for the vector and shuffle controllers are developed manually using inline assembly coding. For a fair comparison with code run exclusively on the MB, the MB's data and instruction caches are attached to the AXI4 memory; the time taken to transfer, via the DMA engine or the scalar processor, data from an external memory, such as DDR, to the data cache and private memories is the same since all connect to the same shared bus, and use the same clock signal and protocol. However, DMA is much faster in the burst transfer mode and should be used for preloading memories.

Although we exclude the time taken by data transfers in our performance comparison between the MB and the VP, the extra time is counted when data is moved between the cache and private memories. Since the private memories are connected independently to the AXI4 interconnect, all of them are accessible and addressable by both the MB and the DMA engine using low-order interleaving. Both the MB and DMA controller view the four private memories as a big vector memory with a single base address. Low-order interleaving is realized by the MUX block. The MB has access to all locations in the vector memory using its base address and the appropriate offset each time. In our prototype, each private memory is 16 Kbytes, so 64 Kbytes of vector memory are available to store application data.

Two distinct types of vector instructions, without (type 1) and with address (type 2) inclusion, are embedded in the C code run by the MB (i.e., using inline macro calls). The MB runs them as one or two store word (SW) instructions, respectively, targeting the VP's memory-mapped interface. Figure 5 shows how type 1 and type 2 instructions are defined using C functions, and how they are used to create macros that represent VP instructions. The type 1 __ADD instruction only needs 32 bits, whereas the type 2 __VLD (unit-stride load) and __VST (unit-stride store) instructions also carry a 32-bit address. The C code in Figure 5 loads two 16-element vectors from the VM and stores the summation result back in the VM. The C structure that defines a vector contains two unsigned integer fields representing the vector's VL and a pointer to its first element in the VM. The latter field actually contains the offset of the first element which must be added to the base address of the VM.

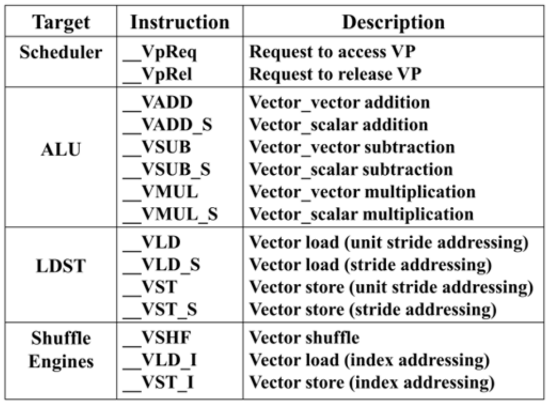
To compile each benchmark written in the C language and containing inline macros for the VP, the MB GNU mb-gcc tool is applied twice, with and without optimization, respectively. Maximum optimization (O3) is invoked that involves inline functioning, loop unrolling and strict aliasing. This optimization causes code rearrangement in order to increase the rate of issuing vector instructions. Eight benchmark algorithms with three alternatives each for the VL value are tested. Therefore, we present results for 24 benchmark instantiations. All the benchmarks are developed using the VP's instruction set architecture (ISA) shown in Figure 6.



```
//functions to define two different kinds of instructions(with and without data)
#define V_instr_a(instr)  asm volatile("sw\t%0,r0"::"d"(instr), "d"(SCHEDULER_ADDRESS));

#define V_instr_b(instr,addr)  asm volatile("sw\t%0,r0""sw\t%1,r0"::"d"(instr),"d"(addr), \
"d"(SCHEDULER_ADDRESS));


//using defined instructions to define vector instructions as macros, X_SHIFT constant
//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(){
__VLD(0,adr1,16,0);    //loading from location adr1 to r0 for VL=16 and thread 0
__VLD(1,adr2,16,0);    //loading from location adr2 to r1 for VL=16 and thread 0
__VADD(2,0,1,16,0);    //r2<=r0+r1 for VL=16 and thread 0
__VST(2,adr3,16,0);    //storing from r2 to location adr3 for VL=16 and thread 0
};
```

Figure 5: Example of C code showing VP instructions implemented as macro calls for vector addition.

The first benchmark is the multiplication of square matrices with size 16*16, 32*32 and 64*64. Three benchmark algorithms are developed for matrix multiplication. Algorithms 1 and 2 calculate one element of the resulting matrix in each loop iteration. Only the location of additions differs between these two algorithms (details follow in the next section). Algorithm 3 improves the vectorization ratio (i.e., ratio of vector to scalar code) since all the elements of a row in the resulting matrix are calculated in each loop iteration. For the sake of comparison, sequential C code for the MB scalar processor is also developed for matrix multiplication that represents the same number of operations with each of these algorithms. Two benchmark algorithms implement FIR (Finite Impulse Response) digital filtering and use the outer product [19]. 16, 32 and 64 tap FIR filters are realized. One of these benchmarks (Algorithm 2, which is presented below), applies special memory initialization to maximize the vectorization ratio in a way that takes advantage of unrolling the loop four times. The next two benchmark algorithms implement FFT using a 16, 32 and 64 point decimation-in-time radix-2 butterfly algorithm [20]. Shuffle instructions are executed in each stage. In each benchmark execution, the results of performing data shuffling by the scalar processor and the shuffle engine, respectively, are observed and compared. More details about the FIR and FFT benchmark variations follow in the next section. The last benchmark is RGB2YIQ (RGB to YIQ color space) mapping. This benchmark, which is the most vectorizable, is run on 16*16, 32*32 and 64*64 pixel matrices. All these benchmark instantiations were also implemented on the MB using sequential code.



Target	Instruction	Description
Scheduler	__VpReq __VpRel	Request to access VP Request to release VP
ALU	__VADD __VADD_S __VSUB __VSUB_S __VMUL __VMUL_S	Vector_vector addition Vector_scalar addition Vector_vector subtraction Vector_scalar subtraction Vector_vector multiplication Vector_scalar multiplication
LDST	__VLD __VLD_S __VST __VST_S	Vector load (unit stride addressing) Vector load (stride addressing) Vector store (unit stride addressing) Vector store (stride addressing)
Shuffle Engines	__VSHF __VLD_I __VST_I	Vector shuffle Vector load (index addressing) Vector store (index addressing)

Figure 6: ISA of the VP.

4. Analysis of Results

Table 2.a and 2.b show results under various execution scenarios for matrix multiplication without compiler optimization and with maximum compiler optimization, respectively. The DMA is used to transfer input data to vector memory. For the sake of simplicity, we show the time taken in each case for producing one element in the resulting matrix. Matrix multiplication Algorithms 1 and 2 multiply a row with a column and add the partial results. Both algorithms use the VP for multiplications. Algorithm 1 uses the MB for the addition of partial products whereas Algorithm 2 uses both the VP and the MB for this purpose. More specifically, partial products are loaded in four vector registers and vector-vector additions are then applied, which are followed by VL/4 MB additions to produce each element in the resulting matrix. Algorithm 3 uses a different technique that produces a single resulting row at a time. More specifically, the MB is only in charge of vector-instruction control flow; all additions and multiplications are done by the VP. This algorithm multiplies a single element of the first matrix with all the 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 by the respective row in the second matrix (element 1 with row 1, element 2 with row 2, and so on) and appropriate partial products are summed up. All the multiplications are performed using scalar-vector multiplication instructions and additions are carried out via vector-vector additions. The results show that by increasing the dimensionality of the matrix and consequently the VL, the time needed to generate one element in the product matrix increases for the element-wise Algorithms 1 and 2 while it decreases slightly for the row-wise Algorithm 3 (due to higher vectorization ratio). Compiler optimization demonstrates the most dramatic beneficial impact on the algorithm that uses the VP the most; this is because the improvement increases faster with the matrix size.

Table 2: Performance comparison for three multiplication algorithms and various VLs. Algorithms 1 and 2 use both the VP and MB. Algorithm 3 uses only the VP. The execution time (in microseconds) is shown for each element produced in the product matrix. (a) Without compiler optimization and (b) with compiler optimization.

Matrix Size Vector Length		Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP+MB)	Algorithm 3 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16*16	VL=16	75	28	4.5	160
32*32	VL=32	144	31	4.36	319
64*64	VL=64	279	34	4.32	630

(a)

Matrix Size Vector Length		Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP+MB)	Algorithm 3 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16*16	VL=16	69	21	1.5	139
32*32	VL=32	136	22	1.43	278
64*64	VL=64	268	23	1.368	552

(b)

Table 3: Performance comparison for FIR filtering with various filter sizes. The times for data exchanges between the global and private memories are included. The times are for calculating all the output elements. (a) Without compiler optimization and (b) with compiler optimization.

Filter Size Vector Length		Input Length	Output Length	Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16 Tap	VL=16	16	32	414	149	4250
32 Tap	VL=32	32	64	1439	278	15615
64 Tap	VL=64	64	128	5331	536	68703

(a)

Filter Size Vector Length		Input Length	Output Length	Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16 Tap	VL=16	16	32	114	52	3813
32 Tap	VL=32	32	64	370	94	14189
64 Tap	VL=64	64	128	1312	178	64102

(b)

Table 3.a and 3.b show performance results for FIR filtering under various scenarios and VLs without and with compiler optimization, respectively. 16, 32 and 64 tap FIR filters are implemented with the size of the input sequence being the same as the size of the filter; therefore, the resulting sequence has double the length of the input. The MB is used to transfer data between the global and vector memories. Two algorithms are developed. Algorithm 1 has no loop unrolling while Algorithm 2 uses special initialization of the vector memory customized for the four lanes by unrolling the loop four times to achieve higher VP utilization. The results in the table include the time to initialize the private memory by transferring data from the global memory. Similar to matrix multiplication, the effect of compiler optimization is more prominent when using only the VP.

Table 4.a and 4.b depict the results for FFT without and with compiler optimization, respectively. We implemented 16, 32 and 64 point FFT using two algorithms. MB is in charge of transferring input and output data between the global and vector memories. In Algorithm 1, the shuffling of data in the vector memory is realized by the MB via the AXI4 interconnect. In Algorithm 2, the distributed data shuffle engines implement shuffling needed in each stage of FFT. As intended, the data shuffle engines which are distributed across the vector lanes have a spectacular beneficial impact on FFT's execution time due to its hefty demand of data shuffling. The relevant speedup is 5.92 and 7.33 for the 64-point FFT without and with compiler optimization, respectively. Furthermore, the 64-point FFT speedup of Algorithm 2 against runs on the MB is 52.07 and 110.45 without and with compiler optimization, respectively. Also, similar to runs of the other benchmarks, the impact of compiler optimization becomes more prominent when the VP utilization increases. In general, the performance advances of our architecture become more manifested with increased VLs in applications.

Table 4: Performance comparison for FFT of various sizes. The execution time includes the overhead of writing and reading between the global and vector memories. The numbers are for calculating all the output results. (a) Without compiler optimization and (b) with compiler optimization.

FFT Size	**Vector Length**	**Algorithm 1 Execution Time(us) (VP+MB)**	**Algorithm 2 Execution Time(us) (VP)**	**Scalar Execution Time(us) (MB)**
16 Point	VL=16	386	132	3850
32 Point	VL=32	814	190	7380
64 Point	VL=64	1783	301	15673

(a)

FFT Size	**Vector Length**	**Algorithm 1 Execution Time(us) (VP+MB)**	**Algorithm 2 Execution Time(us) (VP)**	**Scalar Execution Time(us) (MB)**
16 Point	VL=16	175	42	2520
32 Point	VL=32	379	63	5605
64 Point	VL=64	762	104	11487

(b)

Table 5.a and 5.b show performance results for the RGB2YIO benchmark without and with compiler optimization, respectively. DMA is used to move information about pixels to the vector memory. Although three sizes are chosen for the input pixel array, the results in the table are for calculating the values in a block of 8*8 pixels in the YIQ target space. Since each output pixel value depends only on the corresponding input pixel value, the MB time consumed for calculating an 8*8 block is the same in all three cases. Since this benchmark is extremely vectorizable, the speedup of the VP over the MB is huge. Obviously, it increases even further via compiler optimization.

Table 5: Performance comparison for RGB2YIQ with various VLs. The time is for calculating a block of 8*8 pixels in the YIQ color space. (a) Without compiler optimization and (b) with compiler optimization.

Vector Length	**VP Execution Time(us) (VP)**	**Scalar Execution Time(us) (MB)**
VL=16	63.25	10932
VL=32	31.66	10932
VL=64	16.76	10932

(a)

Vector Length	**VP Execution Time(us) (VP)**	**Scalar Execution Time(us) (MB)**
VL=16	27.58	10572
VL=32	13.87	10572
VL=64	6.99	10572

(b)

To further underscore the need of the VP coprocessor, Figure 7.a shows its speedup compared to MB execution for matrix multiplication using Algorithm 3. The VP achieves a x400 speedup with VL=64 and compiler optimization. A main performance bottleneck in this testbench is the low rate of issuing vector instructions to the VP. Therefore, without optimization the VP is not fully utilized since the time between issuing vector instructions to the VP is larger than the time needed for a vector instruction to be implemented. Using inline assembly-language macros for vector instructions rather than HLL function calls, we reduce this difference as much as possible. We also need to follow two other approaches in the effort to minimize VP idle times. First, increasing the VL of the application algorithm will keep the VP busy for a longer time, hopefully till the next vector instruction is issued. The second approach is to increase the vector instruction issue rate for the VP by applying code optimization. Figure 7.b shows the speedup of the VP+MB system versus the MB for matrix multiplication under Algorithm 2. Two main differences can be observed between the two parts of Figure 7. The rate of speedup improvement for Algorithm 3 with an increasing VL is much higher which implies a higher VP utilization. Also, the effect of compiler optimization is lower for Algorithm 2 because only part of the application is run on the VP.

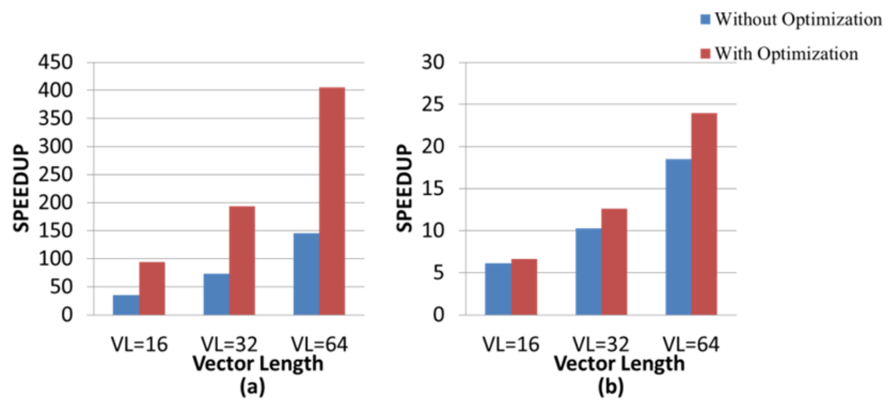


Figure 7: Speedup for matrix multiplication with and without optimization. (a) VP vs. MB for Algorithm 3 and (b) VP+MB vs. MB for Algorithm 2.

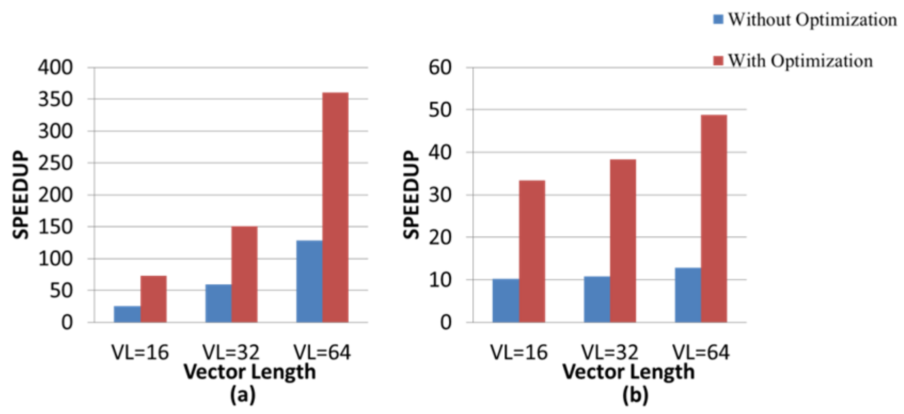


Figure 8: VP vs. MB speedup for FIR filtering with and without optimization. (a) Algorithm 2 and (b) Algorithm 1.

Figure 8 depicts the speedup for FIR filtering under various filter tap sizes and VLs. Figure 8.a presents the speedup of the VP versus the MB for Algorithm 2. For 64 taps with compiler optimization, the speedup is higher than 350 and increases drastically when the VL increases due to higher vectorization ratios. Initializing the private memories of the four lanes helps this process. Figure 8.b shows the speedup for Algorithm 1. Without optimization, the speedup does not keep up with VL increases since this algorithm has a lower vectorization ratio; the MB is involved in each iteration that produces a new element of the result. However, compiler optimization increases the portion of the algorithm that runs on the VP which, in turn, improves acceleration.

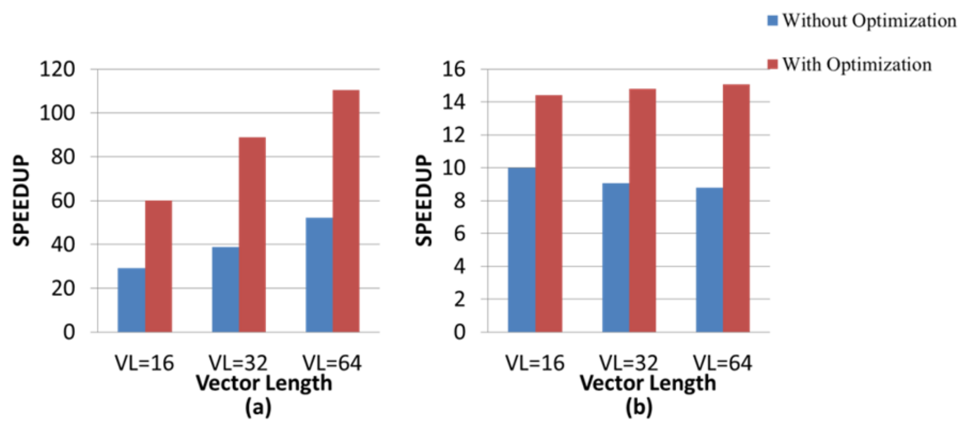


Figure 9: Speedup for FFT. (a) VP with the data shuffle engine vs. MB for Algorithm 2 and (b) VP+MB without the shuffle engine vs. MB for Algorithm 1.

Figure 9.a and 9.b show the FFT speedup for various VLs, with and without the distributed data shuffle engines, respectively. The VP with the data shuffle engines and code optimization achieves a 110-fold speedup over the MB for the 64-point FFT. For Figure 9.b, the data shuffle instructions are implemented by the MB instead of the VP. Without the shuffle engine and without code optimization, the speedup degrades slightly with an increasing VL. Since data shuffling is the most time consuming process in each stage of FFT, performing it on the MB with an increased VL results in slight performance degradation; however, compiler optimization can compensate by increasing the portion run on the VP.

Figure 10 shows the speedup for the highly vectorizable RGB2YIQ benchmark. The VP achieves an impressive 1500-fold speedup over the MB for VL=64 and with compiler optimization. This benchmark approaches the peak performance of the VP since most of the time the VP is fully utilized without waiting for a new vector instruction to be issued.

We also show in Figure 11 the performance/area ratio of the VP over MB execution of the benchmarks for three VL alternatives, assuming maximum compiler optimization. Our benchmarking shows that our VP supports scalability since the ratio generally improves with increases in the VL. Actually, the improvement is faster for a reduced number of data dependencies (i.e., RGB2YIQ), and slower or negligible for a very large number of data dependencies (i.e., FFT).

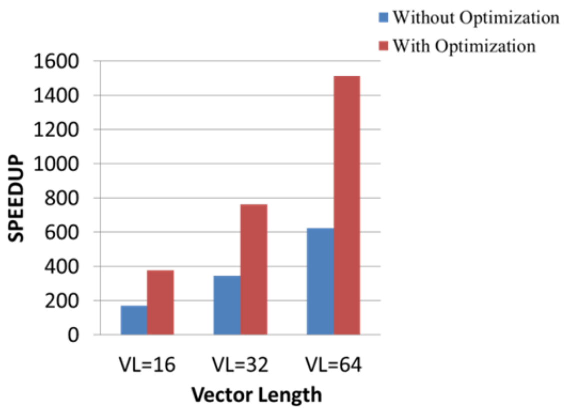


Figure 10: VP vs. MB speedup for RGB2YIQ.

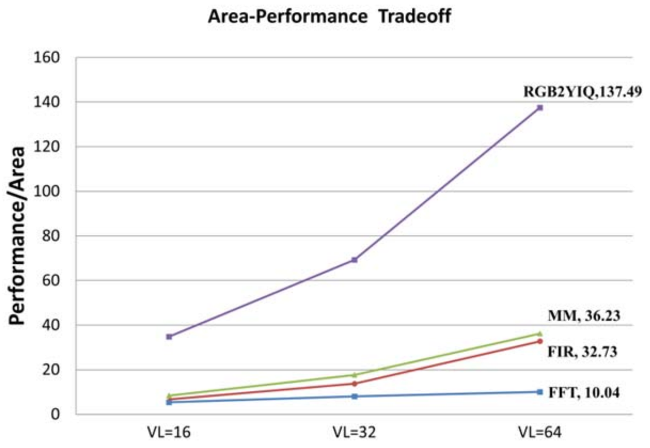


Figure 11: Performance/Area improvement for the VP over the MB.

5. Comparison with Prior Work

For a fair performance comparison with earlier works involving VP designs, we need to count the execution time of applications in clock cycles (e.g., since different target FPGAs used in prototyping support different clock frequencies, etc.). The main bottleneck in benchmarking is the MB core that adds delays to the process of issuing vector instructions to the VP; this decreases the utilization of the VP due to a lesser average density of vector instructions in each lane's ALU and LDST FIFOs. Compiler optimization may ease this problem depending on the application, as discussed in Section 4. For fair VP comparisons independent of compilers and scalar cores, we should target at maximum VP performance. Therefore, all the vector instructions in this section are placed in advance in the VC queue instead of being issued by the MB as encountered in the application code. Also, all private memories are initialized with the needed application data. We thus count the clock cycles really taken by applications on the VP.

To find the minimum number of clock cycles for executing each benchmark, we calculate its total number of instructions. For accuracy, the VP behavior in the case of hazard detection is taken into consideration. Whenever a data hazard is detected by the VC, issuing instructions to the FIFO is stalled and demand for a new vector instruction (VI) is delayed until the corresponding instruction is committed and the pipeline is back to normal operation. Another important issue is the capability of overlapping a data shuffle instruction with subsequent instructions as long as no data hazard is present.

Table 6: Matrix multiplication performance comparison for various VLs.

Vector Length	**TI***	**EFPI****	**#of cycles**	**Stall Rate (%)**	**#of cycles per VI*** **	**Average Utilization (%)**		
						LDST	**ALU**	
VL=16	784	10.44	6893	0	8.79	15.8	29.7	**Ideal**
VL=32	3104	21.11	40461	0	13.05	20.9	42.2	
VL=64	12352	42.44	252441	0	20.48	26.3	51.7	
VL=16	784	10.44	10157	32	12.95	10.7	20.2	**Ideal without private memory**
VL=32	3104	21.11	64845	35	20.89	13.1	26.3	
VL=64	12352	42.44	455309	44	36.86	14.58	28.6	
VL=16	784	10.44	19200	0	24.48	5.7	11.3	**Practical**
VL=32	3104	21.11	73216	0	23.58	11.3	22.3	
VL=64	12352	42.44	280166	0	22.68	23.7	46.6	

*TI=Total Instructions ***VI: Vector Instruction
**EFPI=Effective FLOPs Per Instruction

Table 7: FIR performance comparison for various VLs.

Vector Length	**TI**	**EFPI**	**#of cycles**	**Stall Rate (%)**	**#of cycles per VI**	**Average Utilization (%)**		
						LDST	**ALU**	
VL=16	100	10.24	463	0	4.63	29.3	57.1	**Ideal**
VL=32	196	20.89	1419	0	7.23	37.1	73.2	
VL=64	388	42.22	5341	0	13.76	39.1	77.3	
VL=16	100	10.24	931	50	9.31	14.6	28.4	**Ideal without private memory**
VL=32	196	20.89	3003	52	15.32	17.5	34.6	
VL=64	388	42.22	11691	54	30.13	17.9	35.3	
VL=16	100	10.24	1450	0	14.5	9.3	18.2	**Practical**
VL=32	196	20.89	2850	0	14.54	18.53	36.5	
VL=64	388	42.22	5750	0	14.82	36.1	71.8	

Table 8: FFT performance comparison for various VLs.

Vector Length	**TI**	**EFPI**	**#of cycles**	**Stall Rate (%)**	**#of cycles per VI**	**Average Utilization (%)**			
						LDST	**ALU**	**SHF**	
VL=16	64	10	396	0	6.18	16	40.4	24	**Ideal**
VL=32	80	20	956	0	11.95	16.7	41.7	25.1	
VL=64	96	40	2280	0	23.75	16.8	42	25.2	
VL=16	64	10	672	40	10.5	9.4	23.8	14.1	**Ideal without private memory**
VL=32	80	20	1611	40	20.14	9.9	24.7	14.9	
VL=64	96	40	3820	40	39.79	10	25.1	15	
VL=16	64	10	1300	0	20.31	4.7	12.3	7.4	**Practical**
VL=32	80	20	1550	0	19.37	10.3	26	15.4	
VL=64	96	40	2500	0	26.04	15.4	38.4	23.1	

Table 9: RGB2YIQ performance comparison for various VLs.

Vector Length	TI	EFPI	#of cycles	Stall Rate (%)	#of cycles per VI	Average Utilization (%)		
						LDST	ALU	
VL=16	336	11.42	1437	0	4.27	26.7	66.8	**Ideal**
VL=32	672	22.85	5165	0	7.68	29.7	74.3	
VL=64	1344	45.71	19533	0	14.53	31.4	78.6	
VL=16	336	11.42	2493	42	7.42	15.4	38.5	**Ideal without private memory**
VL=32	672	22.85	9581	46	14.25	16	40	
VL=64	1344	45.71	37581	48	27.96	16.3	42.9	
VL=16	336	11.42	5500	0	16.36	6.9	17.5	**Practical**
VL=32	672	22.85	11100	0	16.51	13.8	34.6	
VL=64	1344	45.71	22400	0	16.66	27.4	68.6	

Tables 6-9 show performance results for executing each benchmark on our VP. “Practical” times are obtained from the already presented results by excluding the times needed to transfer data between the global and private memories. “Ideal” times are obtained by removing any MB delay in issuing instructions to the VP. “Ideal without private memories” times are similar to ideal but, instead of having a private memory in each lane, each lane has access to all memory banks in the vector memory using a crossbar that connects lanes to memories (similar to the architecture in [13]). Under the worst case scenario for vector load and store instructions, only one element per clock cycle can be transferred between the lanes and the vector memory. This, however, is the best case for our VP architecture due to the presence of the private memories that can transfer four elements per clock cycle.

“Total instructions” represents the number of vector instructions issued by the MB, considering all loop iterations. The effective FLOPS per instruction are obtained by dividing the total number of ALU floating-point operations in the benchmark by the total number of instructions. The average number of clock cycles needed per instruction is then calculated for each benchmark. The LDST average utilization of a lane shows the number of vector elements sent to or received from the vector memory in 100 clock cycles. The ALU average utilization represents the number of elements produced by a lane’s ALU in 100 clock cycles. The SHF utilization in Table 8 for the FFT shows the number of elements transferred by the shuffle engines in 100 clock cycles. We can conclude that increasing the VL brings the practical time closer to the ideal one due to higher VP utilization and less idling between instructions issued by the MB.

Table 10: Speedups of our VP and the design in [12] vs. the MB for VL=32.

Architecture	Matrix Multiplication	FIR	FFT
CTS [13], 8 lanes, 1 core	12.97	10.93	49.02
FTS [13], 8 lanes, 2 cores	25.89	21.83	86.76
Our VP, 4 lanes, 1 core	193.5	150.94	88.98

We now compare our 4-lane VP results with those in [13] that assumed a VIRAM-like VP (like us) with eight lanes. A crossbar in the latter design connects the lanes to the global vector memory, and is also used to realize data shuffle and indexed memory instructions. Due to the lack of private memories, there is a high chance of stalls for memory instructions. They used a Xilinx block memory IP (BRAM) for the VRF and their total VRF capacity was 2 Kbytes/lane (ours is 1 Kbyte/lane). Since they used Xilinx IPs to build lane ALUs, many embedded DSP blocks were consumed. The Fast Simplex Link

(FSL) point-to-point interface was employed to connect the VP with the MB. Each of the two MBs in their design uses its own VC and FSL slave and master interfaces to access the VP. To allow the two MB cores to share the VP, three scheduling policies were implemented for their scheduler. In coarse-grain temporal sharing (CTS), the two cores time share the entire VP. In fine-grain temporal sharing (FTS), both cores compete simultaneously for all the VP resources. This is equivalent to simultaneous multithreading (SMT) with respect to VP usage and the two core threads use different vector registers. Vector lane sharing (VLS), finally, assigns distinct lanes to each MB upon demand, as decided by the VC.

As per Table 10, the obtained speedup for the FFT benchmark is smaller than that for FIR filtering and matrix multiplication since the former requires heavy data shuffling. However, our speedups are always higher than those in [12]. This observation becomes even more impressive considering that our VP platform has four, instead of eight, lanes and one core, instead of two. For the FIR benchmark, we compare the speedups of our VP and SODA [4] in reference to their individual host processors (i.e., MB and Alpha, respectively). SODA achieves speedups of up to 19 and 26 for 33- and 65-tap filters, respectively. Our VP accomplishes much higher speedups of 150 and 350 for 32- and 64-tap filters, respectively.

Also, there exist comparative results of FPGA and ASIC realizations involving various designs that make it possible to estimate the relative speedup and improved power dissipation, within an order of magnitude, when an FPGA-based design is moved into the ASIC realm (e.g., [21, 23, 24]). However, an ASIC implementation of our design, a rather hectic and lengthy process, will be a future research objective.

6. Power Analysis

For high accuracy in the estimation of our VP's power and energy consumption, we employ the Xilinx Power Analyzer (XPA) which can determine the power when the activity rate for each signal and net in the hardware are specified. Power dissipation has two major components. Static power dissipation is due to current leaking through the transistors, even without any activities. Dynamic power depends on the design's activities [22].

Estimating the dynamic power of a design requires knowledge of the activity rates for all signals and nets in the hardware. This information is available in the Xilinx SAIF (Switching Activity Interchange Format) and VCD (Value Change Dump) files which are generated after performing timing simulation of the design. We perform timing simulation for each benchmark instantiation using the ISE simulator (ISim) tool and generate the SAIF file that shows the exact activity rates in the placed-and-routed (RAP) design. The design is first synthesized, translated and mapped to the target platform before RAP. The SAIF file is generated between desired intervals during simulation and includes information for the interval. The SAIF, NCD (Native Circuit Description) and PCF (Physical Constraint) files are imported into XPA to obtain accurate estimation of the power consumption before the configuration bit-stream is generated and downloaded to the FPGA.

To find the maximum dynamic energy dissipation, we focus on maximum VP performance. To this end, we focus on that kernel of each benchmark that involves in the calculations the VP, private memories and shuffle engines. The respective vector instructions are then applied directly to the VP instead of being issued by the MB (i.e., there is no delay between consecutive VP instructions). For each kernel, we first perform behavioral simulation to obtain the interval for SAIF generation. The start point is the moment that the first vector instruction from the kernel enters the VC whereas the end point is when the last vector element is written back to the VRF or private memories. After determining the desired interval, the post-RAP simulation is performed for the benchmark and the SAIF file is generated for the desired interval. The device configuration and environmental parameters are set to their default values (e.g., the ambient temperature is 50^0C and the airflow is 250LFM). The static power remains unchanged at 2.878mW for all benchmarks since the same target device is used (FPGAs do not support power gating).

For matrix multiplication with Algorithm 3, which is the most vectorizable in Table 2, the innermost loop that involves three instructions is considered as the target kernel. It is repeated VL times until one row of the product matrix is generated. This kernel includes one load instruction, one vector-scalar multiplication and one vector-vector addition. Table 11 shows the measured values for this benchmark under this maximum power dissipation scenario that assumes no delay between issuing consecutive vector instructions. “Kernel duration” shows the length of the chosen interval. “Application duration” is obtained from the “ideal” numbers in Table 6. It can be seen that the clock distribution network dominates the dynamic power, which is in agreement with earlier results.

Table 11: Power and energy consumption for matrix multiplication.

Vector Length	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		Clock	Signal & Logic	BRAM & IO			
VL=16	550	106.8	71.96	4.16	137	25.06	182.92
VL=32	710	106.8	87.96	5.28	809	161.83	200.04
VL=64	1030	106.8	104.2	6.84	5048	1099.66	217.84

For FIR filtering with Algorithm 2, which is the most vectorizable in Table 3, the target kernel for power estimation is the internal loop that 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 that consist of four load, four vector-scalar multiplication and four vector-vector addition instructions, and maximizes the VP utilization for this benchmark. The power and energy values are presented in Table 12.

Table 12: Power and energy consumption for FIR filtering.

Vector Length	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		Clock	Signal & Logic	BRAM & IO			
VL=16	1150	106.8	93.52	5.36	9.2	1.89	205.68
VL=32	1790	106.8	102.6	6	28.3	6.09	215.4
VL=64	3070	106.8	109.52	6.52	106.8	23.80	222.84

For FFT with Algorithm 2, which is the most vectorizable in Table 4, a single stage of FFT calculation forms the target kernel. Each stage involves sixteen vector instructions which include two data shuffle, six vector-vector multiplication, two load, two store, two vector-vector addition and two vector-vector subtraction instructions. Table 13 contains the results. Since data shuffling does not interfere with internal VP operations, the “ideal” numbers from Table 8 are assumed for the time. It can be seen that when the ratio of ALU to LDST instructions issued increases in the kernel (e.g., FFT), the dynamic power of signals and logic increases since more results are produced in the floating-point computation units.

Table 13: Power and energy consumption for FFT.

Vector Length	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Shuffle Engines+ Crossbar Dynamic Power(mW)	Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		Clock	Signal & Logic	BRAM & IO				
VL=16	1350	106.8	107.36	5.52	22.12	7.9	1.91	241.8
VL=32	2090	106.8	133.12	6.48	22.12	19.1	5.13	268.52
VL=64	3690	106.8	144.16	7	22.12	45.6	12.77	280.08

For the RGB2YIO benchmark, the chosen kernel with the maximum VP utilization converts the color space for one row of the input block. This kernel consists of 21 vector instructions and includes three load, nine scalar-vector multiplication, six vector-vector addition and three store instructions. The "ideal" numbers from Table 9 are used. The results are shown in Table 14.

Table 14: Power and energy consumption for RGB2YIQ.

Vector Length	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		Clock	Signal & Logic	BRAM & IO			
VL=16	2350	106.8	82.96	5.72	28.7	5.6	195.48
VL=32	4230	106.8	93.16	5.64	103.3	21.2	205.2
VL=64	7590	106.8	94.2	5.44	390.6	88.6	206.4

7. Conclusion

We proposed a multi-lane VP architecture as a high-performance coprocessor for data-parallel applications. Assigning a private memory to each vector lane and specifying one set of memory ports exclusively for transactions between that memory and the corresponding lane increases the speedup for memory-based vector instructions. Data shuffling and index addressing are realized using distributed data shuffle engines and a crossbar which is placed between the private and global memories. Benchmarking shows an up to 1500-fold speedup over a scalar processor while the area is increase 11-fold. Detailed performance and power dissipation results for each benchmark are provided. The results prove the viability of our approach.

References

[1] Espasa, R., & Valero, M. (1997, December). Simultaneous multithreaded vector architecture: Merging ILP and DLP for high performance. *4th IEEE International Conference on High-Performance Computing*, pp. 350-357.

[2] Kozyrakis, C. E., & Patterson, D. A. (2003). Scalable, vector processors for embedded systems. *Micro, IEEE*, *23*(6), pp. 36-45.

[3] Kozyrakis, C., & Patterson, D. (2002, November). Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. *35th Annual ACM/IEEE International Symposium on Microarchitecture,* pp. 283-293.

[4] Lin, Y., Lee, H., Woh, M., Harel, Y., Mahlke, S., Mudge, T., & Flautner, K. (2006, June). SODA: a low-power architecture for software radio. *ACM SIGARCH Computer Architecture News*, 34(2), pp. 89-101.

[5] Cho, J., Chang, H., & Sung, W. (2006, May). An FPGA based SIMD processor with a vector memory unit. *IEEE International Symposium on Circuits and Systems*, pp. 525-528.

[6] Yiannacouras, P., Steffan, J. G., & Rose, J. (2008, October). VESPA: portable, scalable, and flexible FPGA-based vector processors. *ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 61-70.

[7] Yu, J., Eagleston, C., Chou, C. H. Y., Perreault, M., & Lemieux, G. (2009). Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems*, 2(2).

[8] Chou, C. H., Severance, A., Brant, A. D., Liu, Z., Sant, S., & Lemieux, G. G. (2011, February). VEGAS: soft vector processor with scratchpad memory. *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 15-24.

[9] Severance, A., Edwards, J., Omidian, H., & Lemieux, G. (2014, February). Soft vector processors with streaming pipelines. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 117-126.

[10] Hagiescu, A., & Wong, W. F. (2011, February). Co-synthesis of FPGA-based application-specific floating point SIMD accelerators. *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 247-256.

[11] Yang, H., & Ziavras, S. G. (2005, September). FPGA-based vector processor for algebraic equation solvers. *IEEE International System on Chip Conference*, pp. 115-116.

[12] Heil, T., Krishna, A., Lindberg, N., Toussi, F., & Vanderwiel, S. (2014). Architecture and performance of the hardware accelerators in IBM's PowerEN processor. *ACM Transactions on Parallel Computing*, *1*(1).

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

[14] Lee, Y., Avizienis, R., Bishara, A., Xia, R., Lockhart, D., Batten, C., & Asanović, K. (2013). Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *ACM Transactions on Computer Systems*, 31(3).

[15] Lo, W. Y., Lun, D. P., Siu, W. C., Wang, W., & Song, J. (2011). Improved SIMD architecture for high performance video processors. *IEEE Transactions on Circuits and Systems for Video Technology*, *21*(12), pp. 1769-1783.

[16] Kennedy, K., & McKinley, K. S. (1992, August). Optimizing for parallelism and data locality. *6th international conference on Supercomputing*, pp. 323-334.

[17] AXI Reference Guide (2012), Xilinx Inc.,http://www.xilinx.com/support/documentation/ ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf.

[18] http://www.opencores.org/projects.

[19] Sung, W., & Mitra, S. K. (1987). Implementation of digital filtering algorithms using pipelined vector processors. *Proceedings of the IEEE*, *75*(9), pp. 1293-1303.

[20] Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, *19*(90), pp. 297-301.

[21] Beldianu, S. F., & Ziavras, S. G. (2014, March). Performance-energy optimizations for shared vector accelerators in multicores. *IEEE Transactions on Computers*, *64*(3), pp. 805-817.

[22] Power Methodology Guide (2011), Xilinx Inc., http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/ug786_PowerMethodology.pdf.

[23] Kuon, I., & Rose, J. (2007, February). Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, *26* (2), pp. 203-215.

[24] Suresh, S., Beldianu, S. F., & Ziavras, S. G. (2013, June). FPGA and ASIC square root designs for high performance and power efficiency. *24th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Washington, D.C., USA, pp. 269-272.