Memory Controller for Vector Processor

Tassadaq Hussain[1,2,3,4], Oscar Palomar[3,4], Osman S. Ünsal[3],
Adrian Cristal[3,4,5], and Eduard Ayguadé[3,4]

[1] Riphah International University, Islamabad, Pakistan
[2] UCERD, Islamabad, Pakistan
[3] Computer Sciences, Barcelona Supercomputing Center , Barcelona, Spain
[4] Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain
[5] Artificial Intelligence Research Institute (IIIA), Centro Superior de Investigaciones Científicas (CSIC), Barcelona, Spain
[1]{first}.{last}@bsc.es , [4]{first name}@ucerd.com

Abstract. To manage power and memory wall affects, the HPC industry supports FPGA reconfigurable accelerators and vector processing cores for data-intensive scientific applications. FPGA based vector accelerators are used to increase the performance of high-performance application kernels. Adding more vector lanes does not affect the performance, if the processor/memory performance gap dominates. In addition if on/off-chip communication time becomes more critical than computation time, causes performance degradation. The system generates multiple delays due to application's irregular data arrangement and complex scheduling scheme. Therefore, just like generic scalar processors, all sets of vector machine – vector supercomputers to vector microprocessors – are required to have data management and access units that improve the on/off-chip bandwidth and hide main memory latency.
In this work, we propose an Advanced Programmable Vector Memory Controller (PVMC), which boosts noncontiguous vector data accesses by integrating descriptors of memory patterns, a specialized on-chip memory, a memory manager in hardware, and multiple DRAM controllers. We implemented and validated the proposed system on an Altera DE4 FPGA board. The PVMC is also integrated with ARM Cortex-A9 processor on Xilinx Zynq All-Programmable System on Chip architecture. We compare the performance of a system with vector and scalar processors without PVMC. When compared with a baseline vector system, the results show that the PVMC system transfers data sets up to 1.40x to 2.12x faster, achieves between 2.01x to 4.53x of speedup for 10 applications and consumes 2.56 to 4.04 times less energy.

1 Introduction

Data Level Parallel (DLP) accelerators such as GPUs [1] and Vectors [2–4], are getting popular due to their high performance per area. New programming environments are making GPU programming easy for general purpose applications. DLP accelerators are very efficient for HPC scientific applications because they can simultaneously process multiple data elements with a single instruction. Due to the reduced number of instructions, the Single Instruction Multiple Data (SIMD) architectures decrease the fetch and

This is a post-peer-review, pre-copyedit version of an article published in:
Hussain, T. [et al.]. Memory controller for vector processor. "Journal of signal processing systems", Novembre 2018, vol. 90, núm. 11, p. 1533-1549.
The final authenticated version is available online at: http://dx.doi.org/10.1007/s11265-016-1215-5

decode bandwidth and exploit DLP for data-intensive applications e.g. matrix & media oriented, etc.

Typically, the vector processor is attached to the cache memory that manages data access instructions. In addition, the vector processors support a wide range of vector memory instructions that can describe different memory access patterns. To access strided and indexed memory patterns the vector processor needs a memory controller that transfers data with high bandwidth. The conventional vector memory unit incurs in delays while transferring data to the vector processor from local memory using a complex crossbar and bringing data into the local memory by reading from DDR SDRAM. To get maximum performance and to maintain the parallelism of HPC applications [5] on vector processors, an efficient memory controller is required that improves the on/off-chip bandwidth and feeds complex data patterns to processing elements by hiding the latency of DDR SDRAM. Such system does not get maximum performance from the accelerator cores due to two major reasons. First, data transfers between main memory and local memory is managed by a separate master (scalar) core due to which almost 65% [6], [7] of the time accelerator core remain idle. Second, the accelerator core needs to access complex memory patterns that add latency while managing local data.

To deal with power issues, the memory wall and the complexity of a system, the HPC industry supports FPGA multi-processor and vector accelerator cores for data-intensive scientific applications. While single-chip hard-core microprocessor on FPGA platforms offers excellent packaging and communication advantages, a softcore approach offers the advantage of flexibility and lower part costs. Many FPGA vendors are now offering such scalar soft processor cores [8] [9] that designers can implement using a standard FPGA. FPGA soft processor cores typically decrease system performance compared with hard-core processors. To alleviate the performance overhead, FPGA based soft vector processors have been proposed [10] [11]. A soft vector processor comprises a parameterized number of vector lanes, a vector memory bank and a crossbar network that shuffles vector operations. Soft vector architecture is very efficient for HPC applications, not only it can process multiple data elements based on a single vector instruction, but also it can reconfigure itself depending upon the required performance.

In this paper, we propose a advanced programmable vector memory controller (PVMC) that efficiently accesses complex memory patterns using a variety of memory access instructions. The PVMC manages memory access patterns in hardware thus improves the system performance by prefetching complex access patterns in parallel with computation and by transferring them to a vector processor without using a complex crossbar network. This allows a PVMC-based vector system to operate at higher clock frequencies. The PVMC includes a *Specialized Memory* unit that holds complex patterns and efficiently accesses, reuses, aligns and feeds data to a vector processor. PVMC supports multiple data buses that increase the local memory bandwidth and reduce on-chip bus switching. The design uses a *Multi DRAM Access Unit* that manages memory accesses of multiple *SDRAM module*s.

We integrated the proposed system with an open source soft vector processor, VESPA [10] and used an Altera Stratix IV 230 FPGA device. We compare the performance of the

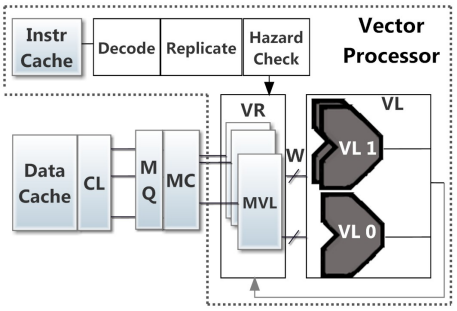

Fig. 1. Generic Vector Processor

system with vector and scalar processors without PVMC. When compared with the baseline vector system, the results show that the PVMC system transfers data sets up to 1.40x to 2.12x faster, achieves between 2.01x to 4.53x of speedup for 10 applications and consumes 2.56 to 4.04 times less energy.

The remainder of the paper is organized as follows. In Section 2 we describe the architecture of a generic vector system. In Section 3 we describe the architecture of PVMC. The Section 4 gives details on the PVMC support for the applications. The hardware integration of PPMC to the Baseline VESPA system is presented in Section 5. A performance and power comparison of PVMC by executing application kernels is given in Section 6. Major differences between our proposal and state of the art are described in Section 7. Section 8 summarizes our main conclusions.

2 Vector Processor

A vector processor is also known as a "single instruction, multiple data" (SIMD) CPU [12], that can operate on an array of data in a pipelined fashion, one element at a time using a single instruction. For higher performance multiple vector lanes (VL) can be used to operate in lock-step on several elements of the vector in parallel. The structure of a vector processor is shown in Figure 1. The number of vector lanes determines the number of ALUs and elements that can be processed in parallel. By default the width (W) of each lane is 32-bit and can be adjusted to 16-bit or 8-bit wide. The maximum vector length (MVL) determines the capacity of the vector register files (RF). Increasing the MVL allows a single vector instruction to encapsulate more parallel operations, but also increases the vector register file size. Larger MVL values allow software to specify greater parallelism in fewer vector instructions. The vector processor uses a scalar core for all control flow, branches, stack, heap, and input/output ports. Performing such processing in the vector processor is not a good use of vector resources, where $1 < \text{MC} < \text{L}$. To reduce the hardware complexity all integer parameters are limited to powers of two.

The vector processor takes instructions from the instruction cache, decodes and proceeds to the replicate pipeline stage. The replicate pipeline stage divides the elements of work, requested by the vector instruction into smaller groups that are mapped onto

the VL lanes. A hazard is generated when two or more of concurrent vector instructions conflict and do not execute in consecutive clock cycles. The hazard check stage examines hazards for the vector and flags register files and stalls if required. Execution occurs in the next two stages (or three stages for multiple instructions) where VL operand pairs are read from the register file and sent to the functional units in the VL lanes.

Modern vector memory units use local memories (cache or scratchpad) and transfer data between the *Main Memory* and the VLs, with parameterized aspect ratio of cache depth, line size, and prefetching. The MC connects each byte in an on-chip memory line to the VL in any given cycle. The vector system has memory instructions for describing consecutive, strided, and indexed memory access patterns. The index memory patterns can be used to perform scatter/gather operations. A scalar core is used to initialize the control registers that hold parameters of vector memory instructions such as the base address or the stride. The vector system includes `vbase, vinc` and `vstride` dedicated registers for memory operations. The `vbase` registers can store any base address which can be auto-incremented by the value stored in the `vinc` registers. The `vstride` registers can store different constant strides for specifying strided memory accesses. For example, if MVL is 16, the instruction `vld.w vector_register, vbase, vstride, vinc` loads the 16 elements starting at `vbase` each separated by `vstride` words, store them in `vector_register`, and finally update `vbase` by adding `vinc` to it. The memory crossbar (MC) is used to route each byte of the cache line (CL) accessed simultaneously to any lane. The MC maps individual memory requests from a VL to the appropriate byte(s) in the CL. The vector memory unit can take requests from each VL and transfers one CL at a time. Several MCs can be used to process memory requests concurrently.

The memory unit of the vector system first computes and loads the requested address in the Memory Queue (MQ) for each lane and then transfers the data to the lanes. If the number of switches in the MC is smaller than the number of lanes, this process will take several cycles. Vector chaining [13] sends the output of a vector instruction to a dependent vector instruction, bypassing the vector register file, thus avoiding serialization, thus allowing multiple dependent vector instructions to execute simultaneously. Vector chaining can be combined with increasing the number of VLs. It requires available functional units; having a large MVL improves the impact on performance of vector chaining. When the loop is vectorized, and the original loop count is larger than the MVL, a technique called strip-mining is applied [14]. The body of the strip-mined vectorized loop operates on blocks of MVL elements. Strip mining transforms a loop into two nested loops: an outer strip-control loop with a step size of multiple of the original loops step size, and an inner loop contains the original step size and the loop body. The vector processor performs strip mining [14] by breaking loops into pieces that fit into vector registers. Strip mining moves vector components of the original loop in the inner loop and transfers all vectorized statements in the body of the outer strip-control loop. In this way, strip mining folds the array-based parallelism to fit in the available hardware. When all MC requests have been satisfied the MQ shifts all its contents up by MC.

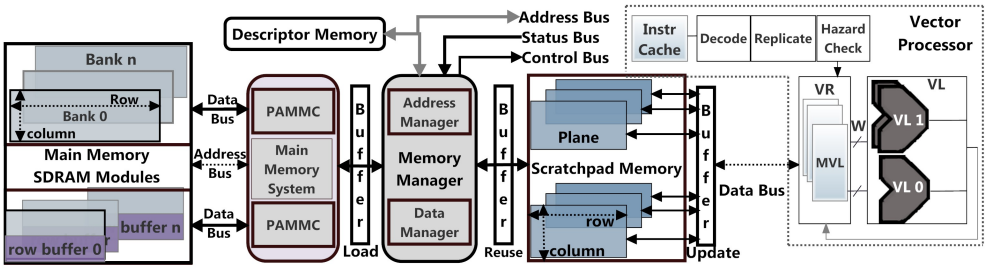

Fig. 2. Advanced Programmable Vector Memory Controller: Vector System

3 Advanced Programmable Vector Memory Controller

The Advanced Programmable Vector Memory Controller (PVMC) architecture is shown on Figure 2, including the interconnection with the vector lanes and the main memory. PVMC is divided into the *Bus System*, the *Memory Hierarchy*, the *Memory Manager* and the *Multi DRAM Access Unit*. The *Bus System* transfers control information, address and data between processing and memory components. The *Memory Hierarchy* includes the *Descriptor Memory*, the *buffer memory*, the *Specialized Memory*, and the *main memory*. The *Descriptor Memory* is used to hold data transfer information while the rest keep data. Depending upon the data transfer the *Address Manager* takes a single or multiple instructions from the *Descriptor Memory* and transfers a complex data set to/from the *Specialized Memory* and *main memory*. The *Data Manager* performs on-chip data alignment and reuse. The *Multi DRAM Access Unit* reads/writes data from/to multiple *SDRAM module*s using several DRAM Controllers.

3.1 Bus System

As the number of processing cores and the capacity of memory components increases the system requires a high-speed bus interconnection network that connects the processor cores and memory modules [15] [16]. The bus system includes the status bus, the control bus, the address bus and the data bus.

The *status bus* holds signals of multiple sources that indicate data transfer requests, acknowledgement, wait/ready and error/ok messages. The bus indicates the current bus operation, and memory read, memory write, or input/output operation. The *control bus* uses signals that control the data movement and carries information of data transfers. The bus is also used to move data between PVMC descriptors and the vector unit's control and scalar registers. The *address bus* is used to identify the locations to read or write data from memory components or processing cores. The *address bus* of the functional units is decoded and arbitrated by the memory manager.

$$Required_{Bandwidth} = Vector_{clock} \times Vector_{Lanes} \times Lane_{width} \quad (1)$$

$$Available_{Bandwidth} = SDRAM_{number} \times Controller_{clock} \times SDRAM_{bus\ width} \quad (2)$$

The *data bus* is used to transmit data to/from *SDRAM module*s. To minimize the data access latency the PVMC scales data bus bandwidth by using multiple data buses, with respect to the performance of the vector core and the capacity of the memory module (SDRAMs). The *required* and *available* bandwidths of the vector system are calculated by using the formulas 1 and 2. Vector$_{clock}$, Vector$_{Lanes}$ and Lane$_{width}$ define the vector system clock, the number of lanes and the width of each lane respectively. SDRAM$_{number}$, Controller$_{clock}$ and SDRAM$_{bus\ width}$ represent the number of separate *SDRAM module*s, the clock speed of each DRAM Controller and the data width of DRAM Controller respectively. To reduce the impact of the memory wall, PVMC uses a separate *data bus* for each *SDRAM module* and local memory (i.e. specialized memory, see Section 3.2.3). To improve bus performance, the *bus clock* can be increased up to a certain extent. The address bus is shared between multiple *SDRAM module*s. The chip select signal is used to enable a specific *SDRAM module*. The control, status and address buses are shared between PVMC and *SDRAM module*s.

3.2 Memory Hierarchy

The PVMC memory hierarchy consists of *Descriptor Memory*, specialized memory, register memory and main memory.

3.2.1 Descriptor Memory The *Descriptor Memory* [17],[18] is used to define data transfer patterns. Each descriptor transfers a strided stream of data. More complex non-contiguous data transfers can be defined by several descriptors. A single *descriptor* is represented by parameters called *Main Memory* address, local memory address, stream, stride and offset. The *Main Memory* and local memory address parameters specify the memory locations to read and write. Stream defines the number of data elements to be transferred. Stride indicates the distance between two consecutive memory addresses of a stream. The offset register field is used to point to the next vector data access through the main address.

3.2.2 Buffer Memory The *Buffer Memory* architecture implements the following features:

- Load/reuse/update to avoid accessing the same data multiple times (*uses the realignment feature*). It handles the increment of the base address, thus reducing loop overhead when applying strip-mining.
- Data realignment when the position in the vector of the reused elements does not match their original position, or when there is a mismatch in the number of elements
- In-order data delivery. In cooperation with the Memory Manager that prefetches data, it ensures that the data of one pattern is sent in-order to the vector lanes. This is used to implement vector chaining from/to vector memory instructions.

The *Buffer Memory* holds three buffers which are the *load buffer*, the *update buffer* and the *reuse buffer*. The *Buffer Memory* transfers data to the vector lanes using the *update buffer*. The *load* and *reuse* buffers are used by the *Memory Manager* that manages

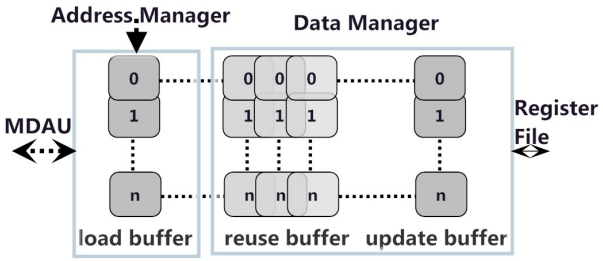

Fig. 3. Data Memory Buffers: Load, Reuse & Update

the *Specialized Memory* data (see Section 3.2.3). For example, if a vector instruction requests data that has been written recently then the *Buffer Memory* performs on-chip data management and arrangement.

3.2.3 Specialized Memory The *Specialized Memory* keeps data close to the vector processor. It has been designed to exploit 2D and 3D data locality. The *Specialized Memory* is further divided into read and write specialized memories. Each *Specialized Memory* provides a data link to the vector lanes. The *Specialized Memory* structure is divided into multiple planes as shown in Figure 2, where each plane hold rows and columns. The row defines the bit-width, and the column defines the density of the plane. In the current configuration, the planes of the *Specialized Memory* have 8- to 32-bit wide data ports, and each data port is connected to a separate lane using the *update buffer*.

The *Specialized Memory* has an address space separated from main memory. PVMC uses special memory-memory operations that transfer data between the *Specialized Memory* and the *main memory*. The data of the read *Specialized Memory* is sent directly to the vector lanes using the *update buffer*, and the results are written back into the write specialized memory.

3.2.4 Main Memory The *Main Memory* has the largest size due to the use of external SDRAM memory modules but also has the highest latency. The *Main Memory* works independently and has multiple *SDRAM module*s. The SDRAM on each memory module implements multiple independent banks that can operate in parallel internally. Each bank represents multiple arrays (rows and column), and it can be accessed in parallel with other banks.

3.3 Memory Manager

The PVMC *Memory Manager* manages the transferring of complex data patterns to/from the vector lanes. The data transfer instructions are placed in the *Descriptor Memory* (see Section 4.5). The *Memory Manager* uses the *Descriptor Memory* to transfer the working set of vector data. The *Memory Manager* is composed of two modules: the Address Manager and the Data Manager.

3.3.1 Address Manager The Address Manager takes a vector transfer instruction and reads the appropriate *Descriptor Memory*. The Address manager uses a single or multiple descriptors, maps addresses in hardware. The Address Manager saves mapped addresses into its address buffer for further reuse and rearranges them.

3.3.2 Data Manager The data manager is used to rearrange the output data of vector lanes for reuse or update. The data memory uses the *reuse*, *update* and *load buffers* (shown in Figure 3) to load, rearrange and write vector data. The `n` shown in Figure 3 is the size of DRAM transfer. When input and output vectors are not aligned the data manager shuffles data between lanes. The data manager reduces the loop overhead by accessing the incremented data and reuses previous data when possible. For example, if the increment is equal to 1 the data manager shifts one data element and requests one element to load from the main memory. The incremented address is managed by the address and data managers that align vector data if required.

The *Memory Manager* takes memory address requests from the control bus and the *Address Manager* reads the data transfer information from the *Descriptor Memory* . The *Data Manager* checks data requests from the specialized memory, if data is available there then the data manager transfers it to the *update buffer*. If the data requests are not available then, the Memory Manager transfers the data request information to the *Multi DRAM Access Unit* (see 3.4) which loads data to the *load buffer*. The *load buffer* along with the *reuse buffer* perform data alignment and reuse where required, and fill the *update buffer*. The *update buffer* transfers data to the vector lanes.

3.4 Multi DRAM Access Unit

The *Multi DRAM Access Unit* (MDAU) accesses data from the *main memory*. The *Main Memory* organization is classified into data, address, control, and chip-select buses. The data bus that transmits data to and from the *Main Memory* is 64 bits wide. A shared address bus carries row, column and bank addresses to the main memory. There is a chip-select network that connects the *MDAU* to each *SDRAM module*. Each bit of chip select operates a separate *SDRAM module*.

MDAU can integrate multiple *DRAM Controller*s using separate data buses, which increases the memory bandwidth. There is one *DRAM Controller* per *SDRAM module*. In the current evaluation environment, two *DRAM Controller*s are integrated. Each *DRAM Controller* takes memory addresses from the *Memory Manager*, performs address mapping from physical address to DRAM address and reads/writes data from/to its *SDRAM module*.

4 PVMC Functionality

In this section, we discuss the important challenges faced by the memory unit of soft vector processors and explain our solution. The section is further divided into five subsections: *Memory Hierarchy*, *Memory Crossbars*, *Address Registers*, *Main Memory Controller* and *Programming Vector Accesses*.

4.1 Memory Hierarchy

A conventional soft vector system uses the cache hierarchy to improve the data locality by providing and reusing the required data set to functional units. With a high number of vector lanes with strider data transfers, the vector memory unit does not satisfy the data spatial locality. PVMC improves the data spatial locality by accessing more data elements than MVL into its *Specialized Memory* and transferring them using the *buffer memory*. Non-unit stride accesses do not exploit spatial locality offered by caches resulting in a considerable waste of resources. PVMC manages non-unit stride memory accesses similar to unit-stride ones. Like a cache of soft vector processor, the PVMC *Specialized Memory* temporarily holds data to speed up later accesses. Unlike a cache, data is deliberately placed in the *Specialized Memory* at a known location, rather than automatically cached according to a fixed hardware policy. The PVMC *memory manager* along with the *Buffer Memory* hold information of unit and non-unit strided accesses, update and reuse them for future accesses.

4.2 Memory Crossbars

To load and store data in a conventional vector system, the vector register file is connected to the data cache through separate read and write crossbars. When the input to the vector lanes is mismatched the vector processor needs an extra instruction that moves and aligns the vector data. The crossbars shuffle bytes/halfwords/words from their byte-offset in memory into word size at a new byte-offset in the vector register file. The size of the crossbars is constrained on one end by the overall width of the vector register file, and on the other side by the overall width of the on-chip memory/cache. The size and complexity of crossbars grow when the vector processor is configured to implement more lanes. The PVMC uses the *Buffer Memory* to transfer data to the vector register file which is simpler than using crossbar and data alignment. The *Buffer Memory* aligns data when output vector elements are needed to process with new input elements. It also reuses and updates existing vector data and loads data which is not present in the *Specialized Memory*.

4.3 Address Registers

The vector processor uses address registers to access data from *main memory*. The memory unit uses address registers to compute the effective address of the operand in *main memory*. A conventional vector processor supports unit-stride, strided and indexed accesses. In our current evaluation environment, the PVMC system uses a separate register file to program the *Descriptor Memory* using data transfer instructions to comply with the MIPS ISA. The PVMC *Descriptor Memory* can perform accesses longer than the MVL without modifying the instruction set architecture. PVMC uses a single or multiple descriptors to transfer various complex non-stride accesses.

4.4 Main Memory Controller

The conventional Main Memory Controller (MMC) uses a direct memory access (DMA) or Load/Store unit to transfer data between *Main Memory* and cache memory. Thus,

the vector memory unit uses a single DMA request to transfer a unit-stride access between *Main Memory* and a cache line. But for complex or non-unit strided accesses the memory unit uses multiple DMA or Load/Store requests which require extra time to initialize addresses, synchronise on-chip buses and SDRAMs. The PVMC MDAU uses descriptors for unit and non-unit stride accesses which improve the memory bandwidth by transferring descriptors to the memory controllers, rather than individual references and by accessing data from multi-SDRAM devices.

4.5 Programming Vector Accesses

Figures 4 (a), (b) and (c) show vector loops (with MVL of 64) for a scalar processor architecture, conventional vector architecture and the PVMC, including the PVMC memory transfer instructions respectively. The `VLD.S` instruction transfers data with the specified stride from *Main Memory* to vector registers using cache memory. For long vector access and a high number of vector lanes, the memory unit generates delay when data transfers do not fit in a cache line. This also requires complex crossbars

```
for( i = 0; i<length; i = i+1 )
        {
                data_out[i] = a [ i * 64 ] + b [ i ] + c [ i ];
        }
```

(a)

```
/*Address of data_out    = 0x10000000 */
/*Address of a           = 0x00000000 */
/*Address of b           = 0x00000100 */
/*Address of c           = 0x00000200 */

for(i=0; i<length; i+=64)
    {
        VLD.S (/*Main Memory*/ 0x00000000+i, /* Vector Register*/, VR0, /*Stride*/ 0x40);
        VLD (/*Main Memory*/ 0x00000100+i, /* Vector Register*/, VR1);
        VADD VR0, VR1, VR2
        VLD (/*Main Memory*/ 0x00000200+i, /* Vector Register*/, VR3);
        VADD VR2, VR3, VR3
        VST (/* Main Memory*/ 0x10000000+i,/*Vector Register*/ VR3);
    }
```

(b)

```
PVMC_VLD (/*Main Memory*/ 0x00000000, /*Local Memory */ 0x00001000, /*Size*/ length,  /*Stride*/ 0x40 );
PVMC_VLD (/*Main Memory*/ 0x00000100, /*Local Memory */ 0x00001040, /*Size*/ length,  /*Stride*/ 0x04 );
PVMC_VLD (/*Main Memory*/ 0x00000200, /*Local Memory */ 0x00001080, /*Size*/ length,  /*Stride*/ 0x04 );
    for(i=0; i<length; i+=64)
        {
            VLD  (/*Local Memory*/ 0x00001000+i, /* Vector Register*/, VR0);
            VLD  (/*Local Memory*/ 0x00001040+i, /* Vector Register*/, VR1);
            VADD VR0, VR1, VR2
            VLD  (/*Local Memory*/ 0x00001080+i, /* Vector Register*/, VR3);
            VADD VR2, VR3, VR3
            VST (/* Local Memory*/ 0x11000000+i, /*Vector Register*/ VR3);
        }
PVMC_VST (/*Main Memory*/ 0x10000000, /*Local Memory */ 0x11000000, /*Size*/ length,  /*Stride*/ 0x04 );
```

(c)

Fig. 4. (a) Scalar Loop (b) Vector Loop (c) PVMC Vector Loop

and efficient prefetching support. Delay and power increase for complex non-stride accesses and crossbars. The `PVMC_VLD` instruction uses a single or multiple descriptors to transfer data from the *Main Memory* to the *Specialized Memory*. PVMC rearranges and manages accessed data in the *Buffer Memory* and transfers it to vector registers. In Figure 5, PVMC prefetches vectors longer than MVL in the *Specialized Memory*. After completing the first transfer of MVL, the PVMC sends a signal to the vector processor that acknowledges that the register elements are available for processing. In this way PVMC pipelines the data transfers and parallelizes computation, address management and data transfers.

A common concern, when using soft vector processors, is compiler support. A soft core vector processor typically requires in-line assembly code that translates vector instructions with a modified GNU assembler. In order to describe how PVMC is used, the supported memory access patterns are discussed in this section. We provide C macros which ease the programming of common access patterns through a set of function calls, integrated with an API. The memory access information is included in the PVMC header file and provides function calls (e.g. STRIDED(), INDEXED(), etc.) that require basic information of the local memory and the data set. The programmer has to annotate the code using PVMC function calls. The function calls are used to transfer the complete data set between *Main Memory* and *Specialized Memory*. PVMC supports complex data access patterns such as strided vector accesses and transfers complex data patterns in parallel with vector execution.

For multiple or complex vector accesses, PVMC prefetches data using vector access function calls (e.g. INDEXED(), etc.), arranges them according to the predefined patterns and buffers them in the *Specialized Memory*. The PVMC memory manager efficiently transfers data with long strides, longer than MVL size and feeds it to a vector processor. For example, a 3D stencil access requires three descriptors. Each descriptor accesses a separate (*x, y* and *z*) vector in a different dimension, as shown in Figure 6. By combining these descriptors, the PVMC exchanges 3D data between the *Main Memory* and the *Specialized Memory* buffer. The values X, Y and Z define the width (row_size), height (column_size) and length (plane_size) respectively of the 3D memory block. When n=4, 25 points are required to compute one central point. The 3D-Stencil has x, z and y vectors having direction of row, column and plane respectively. The x, y and z vectors have length of 8, 9 and 8 points respectively. The vector *x* has unit stride, the vector *z* has stride equal to *row_size* and the vector *y* has stride equal to the size of one plane, i.e. $row\_size \times column\_size$.

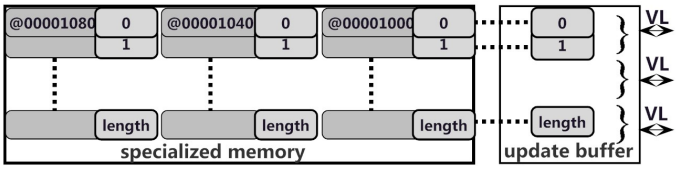

Fig. 5. PVMC Data Transfer Example

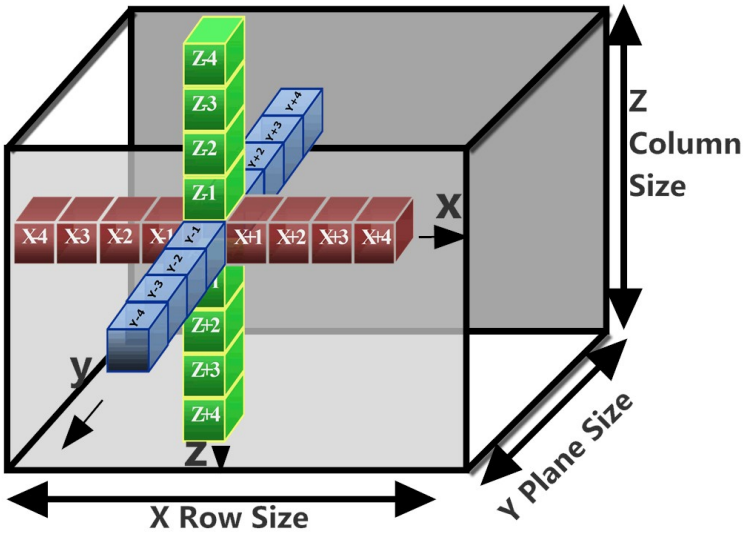

Fig. 6. 3D Stencil Vector Access

5 Experimental Framework

In this section, we describe the PVMC and VESPA vector systems as well as the Nios scalar system. The Altera Quartus II version 13.0 and the Nios II Integrated Development Environment (IDE) are used to develop the systems. The systems are tested on an Altera Stratix-IV FPGA based DE4 board. The section is further divided into three subsections: the *VESPA system*, the *PVMC system* and the *Nios system*.

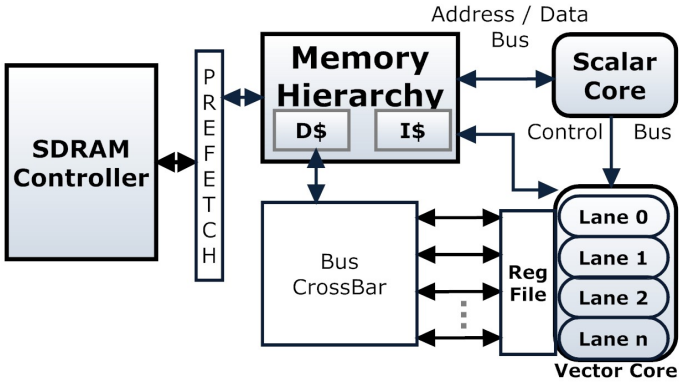

Fig. 7. Baseline VESPA System

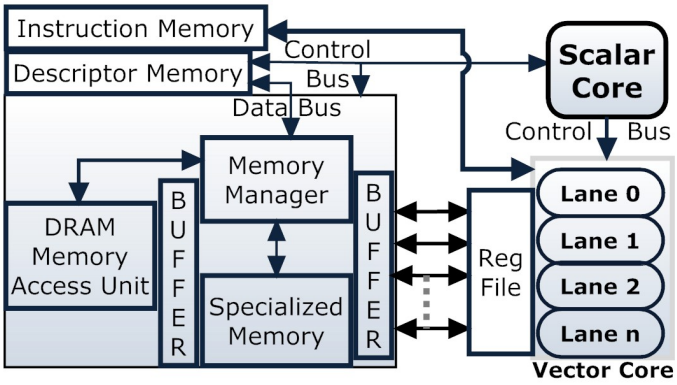

Fig. 8. PVMC System

5.1 The VESPA System

The FPGA based vector system is shown in Figure 7. The system architecture is further divided into the *Scalar core*, the *Vector core* and *Memory System*.

5.1.1 Scalar Core An SPREE [19] scalar processor is used to program the VESPA system and perform scalar operations. The SPREE is a 3-stage MIPS pipeline with full forwarding core and has a 4K-bit branch history table for branch prediction. The SPREE core keeps working in parallel with the vector processor with the exception of control instructions and scalar load/store instructions between the two cores.

5.1.2 Vector Core A soft vector processor called VESPA (Vector Extended Soft Processor Architecture) [10] is used in the design. VESPA is a parameterizable design enabling a large design space of possible vector processor configurations. These parameters can modify the VESPA compute architecture, instruction set architecture, and memory system. The vector core uses a MVL of 128.

5.1.3 Memory System The baseline VESPA vector memory unit (shown in Figure 7) includes an SDRAM controller, cache and bus crossbar units. The SDRAM controller transfers data from *Main Memory* (*SDRAM modules*) to the local cache memory. The Vector core can access only one cache line at a time that is determined by the requesting lane with the lowest lane identification number. Each byte in the accessed cache line can be simultaneously routed to any lane through the bus crossbar. Two crossbars are used, one read crossbar and one write crossbar.

5.2 The Proposed PVMC System

The PVMC based vector system is described in Sections 2 and 3 and shown in Figure 8. The major difference between the PVMC and VESPA systems is the memory system.

Table 1. Brief description of application kernels

Application	Description	Access Pattern
FIR	Calculates the weighted sum of the current and past inputs	Streaming
1D Filter	Low pass 1D Filter	1D Block
Tri-Diogonal	Determining optimal local alignments between nucleotide or protein sequences	Diagonal
Mat_Mul	Performs Matrix Multiplication Output= Row[Vector] Column[Vector] X=YZ	Row & Column
RGB2CMYK	Converts RGB image data into CMYK format	1D Block
RGB2Gray	Converts 24bit RGB to 8 bit Gray scale image	1D Block
Gaussian	Applies discrete convolution filter to approximate the second order derivatives	2D Block
Image Blend	Blend two images and generate one image file	2D Tiling
K-Mean	A method of vector quantization, perform cluster analysis in data mining.	Strided 1D
3D-Stencil	An algorithm that averages nearest neighbor points (size 8x9x8) in 3D	3D Stencil

The PVMC system manages on-chip data and off-chip data movement using the *Buffer Memory* and the *Descriptor Memory*. The memory crossbar is replaced with the *Buffer Memory* which rearranges and transfers data to the vector lanes. The *Specialized Memory* is used instead of a cache memory, that handles complex patterns and transfers them to the vector lanes with less delay.

5.3 The Baseline Nios System

The Nios II scalar processor [9] is a 32-bit embedded-processor architecture designed specifically for the Altera family of FPGAs. The Nios II architecture is an RISC soft-core architecture which is implemented entirely in the programmable logic and memory blocks of Altera FPGAs. Two types of systems having different Nios cores are used; the Nios II/e and the Nios II/f. The Nios II/e system is used to achieve the smallest possible design consuming less FPGA logic and memory resources. The core does not support caches and saves logic by allowing only one instruction to be in-flight at any given time which eliminates the need for data forwarding and branch prediction logic. The Nios II/f system has a fast Nios processor for high performance that implements a barrel shifter with hardware multipliers, branch prediction and 32Kbyte Data and Instruction caches. An Altera Scatter-Gather DMA (SG-DMA) along with SDRAM controller is used that handles multiple data transfers efficiently.

5.4 Applications

Table 1 shows the application kernels which are executed on the vector systems along with their memory access patterns. The set of applications covers a wide range of patterns allowing us to measure the behaviour and performance of data management and data transfer of the systems in a variety of scenarios.

6 Results and Discussion

In this section, the resources used by the memory and bus systems, the application performance, the dynamic power and energy and the memory bandwidth of the PVMC vector system are compared with the results of the non-PVMC vector system and the baseline scalar systems.

6.1 Memory & Bus System

Multiple memory hierarchies and different bus system configurations of PVMC & VESPA systems are compiled using Quartus II to measure their resource usage, maximum operating frequency and leakage power.

Table 2 (a) presents the maximum frequency of the memory system for 1 to 64 vector lanes with 32kB of cache/specialized memory. The VESPA system uses crossbars to connect each byte of the cache line to the vector lanes. Increasing the number of lanes requires more crossbars and a larger multiplexer that routes data between vector lanes and cache lines. This decreases the operating frequency of the system. For the VESPA vector processor, results show that increasing the number of vector lanes from 1 to 64 requires larger crossbar multiplexer switches that forces to operate at lower frequency. The PVMC *Specialized Memory* uses separate read and write specialized memories that reduce the switching bottleneck. The vector lanes read data from read *Specialized Memory* for processing and transfer it back to the write specialized memory. The on-chip data alignment and management is done by the *Data Manager* and the *buffer memory*. This direct coupling of the *Specialized Memory* and vector lanes using the *update buffer* is very efficient and allows the system to operate at a higher clock frequency. Table 2 (b) presents the maximum frequency for the data bus to operate multiple memory controllers. The PVMC data bus supports a dedicated bus for each SDRAM controller which increases the bandwidth of the system. The data bus of VESPA system supports only a single SDRAM controller.

Table 3 shows the resource utilization of the memory hierarchy of the VESPA and PVMC systems. The memory hierarchy is compiled for 64 lanes with 32KB of memory and several line sizes. Column *Line Size* presents cache line and *update buffer* size in bytes of the VESPA and PVMC systems respectively. The VESPA system cache memory uses cache lines to transfer each byte to the vector lanes. The PVMC *update buffer* is managed by the *data manager* and is used to transfer data to the vector lanes. Column *Reg, LUT* shows the resources used by the cache controller and the memory manager of the VESPA and PVMC systems respectively. Column *Memory Bits* presents the number of BRAM bits for the local memory. The PVMC memory system uses separate read

Vector Lanes	1	2	4	16	32	64
VESPA fmax	142	130	125	115	114	110
PVMC fmax	195	187	187	185	182	180

(a)

Sys Bus	1 Layer	2 Layer
VESPA	157	-
PVMC	292	282

(b)

Table 2. (a) Local Bus Maximum Frequency (MHz) (b) Global Bus Maximum Frequency (MHz)

Table 3. Resource Utilization of the Memory Hierarchy

	Local Memory 32KB			Main Memory		Leakage
	Line Size	Reg, LUT	Memory Bits	Controller	Reg, LUTs	Power
VESPA	128	1489, 3465	304400	1	2271, 1366	1.02
	256	1499, 5529	305632	1	2271, 1366	1.15
PVMC	128	90, 1030	613134	1	1742, 1249	0.70
	256	108, 1047	615228	2	3342, 2449	0.80

and writes specialized memories, and therefore, it occupies twice the number of BRAM bits. The data manager of the PVMC memory system occupies 3 to 5 times fewer resources than the VESPA memory system. Column *Main Memory* presents the resource utilization of the DRAM controllers. The VESPA system does not support two SDRAM controllers. Column *Power* shows leakage power in watts for the VESPA and PVMC memory systems. The leakage current of the VESPA system is higher than in PVMC, because it requires a complex crossbar network to transfer data between the cache and the vector lanes and requires more multiplexers.

6.2 Performance Comparison

For performance comparisons, we use the applications of Table 1. We run the applications on the Nios II/e, Nios II/f and VESPA systems and compare their performance with the proposed PVMC vector system. Nios II/e, VESPA and PVMC systems run at 100 MHZ. The VESPA and the PVMC systems are compiled using 64 lanes with 32kB of cache and *Specialized Memory* respectively. The Nios II/f system operates at 200 Mhz using data and instruction caches of 32KB each. All systems use a single SDRAM controller to access the main memory.

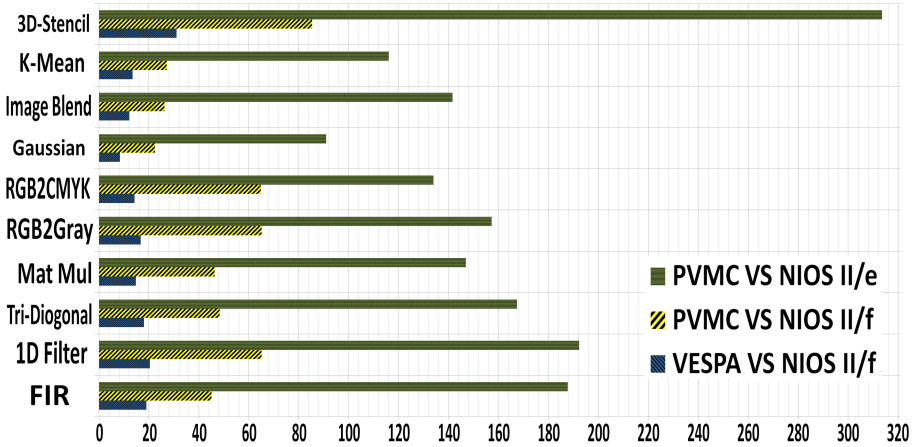

Fig. 9. Speedup of PVMC and VESPA over Nios II/e and Nios II/f

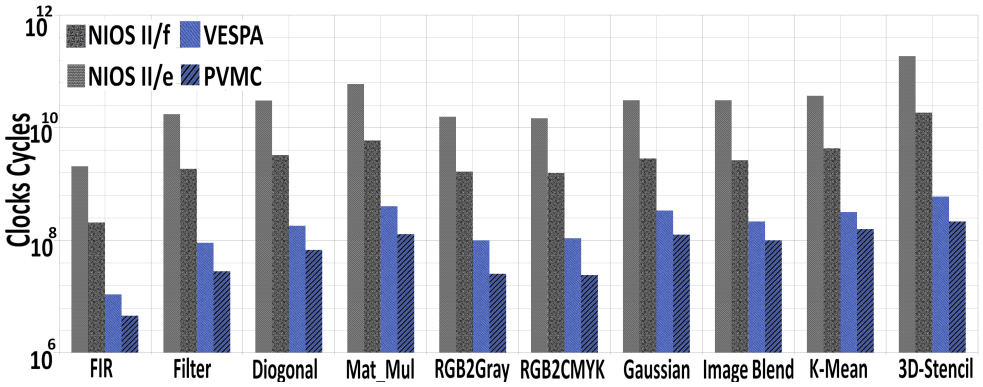

Fig. 10. Vector and Scalar Systems: Application Kernels Execution Clocks

Figure 9 shows the speedups of VESPA and PVMC systems over Nios II/f and Nios II/e. Results show that vector execution with the PVMC is 8.3x and 31.04x faster than the Nios II/f. When compared with the Nios II/e, the PVMC improves speed between 90x and 313x which shows the potential of vector accelerators for high performance.

In order to discard that the speedups over the scalar processor NIOS are caused by using SPREE as the scalar unit of the vector processor, we execute FIR, Mat_Mul and 3D-Stencil application kernels on a SPREE scalar processor, i.e. with the vector processor disabled. While comparing performance of FIR, Mat_Mul and 3D-Stencil kernels on SPREE, Nios II/e and Nios II/f scalar processors, the results show that SPREE improves speed between 5.2x and 8.6x over Nios II/e, whereas against Nios II/f the SPREE is not efficient. The Nios II/f achieves speedups between 1.27x and 1.67x over the SPREE scalar processor. The results show that Nios II/f performs better than Nios II/e and SPREE scalar processors.

Figure 10 shows the execution time (clock cycles) for the application kernels. The X and Y axis represent application kernels and number of clock cycles, respectively. Each

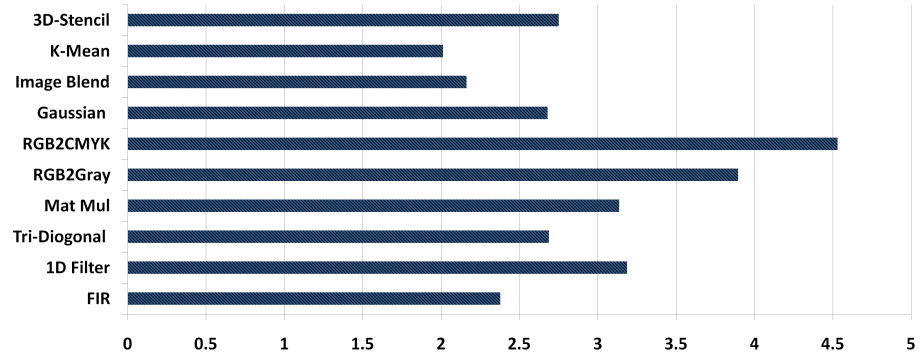

Fig. 11. Speedup of PVMC over VESPA

System @MHz	Lanes	Reg, LUTs	Dynamic Power and Energy			
			FPGA Core	SDRAM	Total	Energy
Nios II/e @100		7034 , 7986	1.47	1.76	3.23	581.17
Nios II/e @200		8612 , 8076	1.65	2.26	3.91	342.46
Nios ll/f @100		9744 , 10126	2.09	1.67	3.76	82.56
Nios ll/f @200		12272 , 10256	3.11	2.51	5.82	48.38
VESPA @100	1	7227 , 7878	1.54	2.24	3.78	101.99
	4	7867 , 12193	1.87	2.24	4.17	60.04
	16	10090 , 31081	3.19	2.24	5.35	14.36
	32	13273 , 57878	4.67	2.24	6.29	9.42
	64	19641 , 103857	5.57	2.25	7.78	7.03
PVMC @100	1	5227 , 5587	1.10	2.11	3.23	39.85
	4	5856 , 6193	1.30	2.11	3.51	20.08
	16	8817 , 21261	1.91	2.11	4.32	4.16
	32	10561 , 45658	2.86	2.11	4.97	2.54
	64	15564 , 88934	4.01	2.11	6.13	1.75

Table 4. Systems: Resource, Power and Energy utilization

bar represents the application kernel's computation time and memory access time in logarithmic scale (less is better). Figure 11 presents the speedup of PVMC over VESPA. By using the PVMC system, the results show that the FIR kernel achieves 2.37x of speedup over VESPA. The application kernel has streaming data accesses and requires a single descriptor to access a stream that reduces the address generation/management time and on-chip request/grant time. The 1D Filter accesses a 1D block of data and achieves 3.18x of speedup. The Tri-diagonal kernel processes the matrix with sparse data placed in diagonal format. The application kernel has a diagonal access pattern and attains 2.68x of speedup. The Mat_Mul kernel accesses row and column vectors. PVMC uses two descriptors to access the two vectors. The row vector descriptor has unit stride whereas the column vector has a stride equal to the size of a row. The application yields 3.13x of speedup. RGB2CMYK and RGB2Gray take 1D block of data and achieve 3.89x and 4.53x of speedup respectively. The Motion Estimation and Gaussian applications take 2D block of data and achieve 2.67x and 2.16x of speedup respectively. The PVMC system manages addresses of row and column vectors in hardware. The 3D-Stencil data uses row, column and plane vectors and achieves 2.7x of speedup. The K-Mean kernel has 1D strided and load/store accesses the kernel achieves 2.01x of speedup. The vectorized 3D-stencil code for VESPA always uses the whole MVL and unit-stride accesses and accesses vector data by using vector address registers and vector load/store operations. The VESPA system multi-banking methodology requires a larger crossbar that routes requests from load/store units to cache banks and another one from banks back to ports. This also increases the cache access time but reduces the simultaneous read and write conflicts.

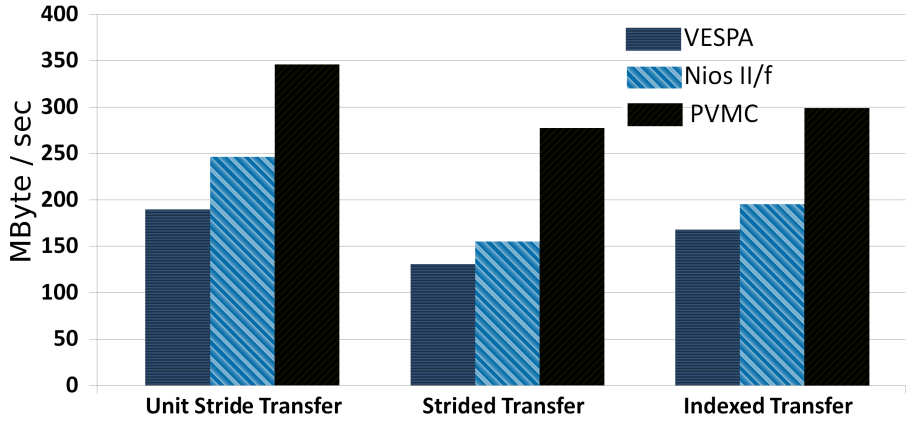

Fig. 12. Vector & Scalar Systems: Memory Bandwidth

6.3 Dynamic Power & Energy

To measure voltage and current the DE4 board provides a resistor to sense current/-voltage and 8-channel differential 24-bit analog to digital converters. Table 4 presents dynamic power and energy of different systems using a filter application kernel with 2M Byte of input data set, 1D block (64 elements) of data access and 127 arithmetic operations on each block of data. Column System@MHz shows the operating frequency of the Nios II/e and Nios II/f cores and the VESPA and PVMC systems. The vector cores execute the application kernel using different numbers of lanes while the clock frequency is fixed to 100 MHz. To control the clock frequencies all systems use a single phase-locked loop (PLL). Columns *Reg, LUTs* and *Mem Bits* show the amount of logic and memory in bits respectively utilized by each system. The Nios II/e does not have a cache memory and only uses program memory. Column *Dynamic Power and Energy* presents run time measured power of scalar and vector systems while executing the filter application kernel and calculated energy for power and execution time. Column FPGA Core includes the power consumed by on-chip FPGA resources and PLL power. Column SDRAM power presents the power of the SDRAM memory device. The power of Nios II/e and Nios II/f increases with frequency. Results show that the PVMC draws 21.2% less power and 4.04x less energy than the VESPA system, both using 64 lanes. For a single lane configuration, PVMC consumes 14.55% less power and 2.56x less energy. This shows that PVMC improves system performance and handles data more efficiently results improve with a higher number of lanes. The PVMC using a single lane since/and operating at 100 MHz draws 14%, 44% less power and 14.5x, 8.5x less energy than a Nios II/f core operating at 100 MHz and 200 MHz respectively. Whereas, when compared to a Nios II/e core at 100 MHz and 200 MHz, the PVMC system draws .03% and 17.3% less power respectively and consumes and 2.07x, 1.21x times less energy.

6.4 Bandwidth

In this section, we measure the bandwidth of the PVMC, VESPA and Nios ll/f systems by reading and writing memory patterns. The PVMC with a single SDRAM controller is also executed on a Xilinx Virtex-5 ML505 FPGA board, and results are very similar. The systems have 32 bit on-chip data bus operating at 100 MHZ that provides a maximum bandwidth of 400 MB. The Nios II/f uses the SGDMA controller and uses multiple load/store or Scatter/Gather DMA calls to transfer data patterns. The PVMC can achieve maximum bandwidth by using a data transfer size equal to the data set. In order to check the effects of memory and address management units over the system bandwidth, we transfer data between processor and memory using three types of transfers: the *unit stride transfer*, *strided transfer*, and *indexed transfer* (scatter/gather). The X-axis (shown in Figure 12) presents three types of data transfers. Each data transfer reads and writes a data set of 2MB from/to the SDRAM. All three transfers have a transfer size of 1024B. The type *unit stride transfer* and *strided transfer* contain a unit and 64B strides between two consecutive elements respectively. The type *indexed transfer* merges non-contiguous memory accesses to a continuous address space. The *indexed transfer* reads a series of *unit stride transfer* instructions that specify the data to be transferred. For the data of *unit stride transfer* type, results show that PVMC transfers data 1.82x and 1.40x faster than VESPA and Nios II/f respectively. While transferring data with the *strided transfer* type, PVMC improves bandwidth 2.12 and 1.79 times. Results show that PVMC improves bandwidth up to 1.78x and 1.53x for *indexed transfer*. Nios II/f uses the SGDMA controller that handles transfer in parallel. SGDMA follows the bus protocol and requires a processor that provides data transfer instructions. For *indexed transfer*, Nios II/f uses multiple instructions to initialize SGDMA. SGDMA can begin a new transfer before the previous data transfer completes with a delay called pipeline latency. The pipeline latency increases with the number of data transfers. Each Data Transfer requires bus arbitration, address generation and SDRAM bank/row management. The VESPA hardware prefetcher transfers data to vector core using cache memory. The cache memory streams all cache lines, including the prefetched lines, to the vector core using the memory crossbar. The PVMC data transfers uses few descriptors that reduce run-time address generation and address request/grant delay and improve bandwidth by managing addresses at compile-time and by accessing data from multi-DRAM devices and multi-banks in parallel.

6.5 ARM Cortex-A9 Evaluation System Performance

The PVMC is also integrated with the ARM processor architecture and tested on Xilinx Zynq All-Programmable System on Chip (SoC) [20] Development platform. Vector multiplication applications are executed on NEON coprocessor of the *ARM Cortex-A9 Evaluation System*s. The NEON coprocessor uses SIMD instruction sets to process vector multiplication. *ARM Cortex-A9 Evaluation System* architecture uses single NEON coprocessor having 512 bytes of register file, 64-bit register size and perform single precision floating point instructions on all lanes. The ARM Generic Memory Conroller (ARM-GMC) uses 512 MB of DDR3 *Main Memory* with 1 Gbps bandwidth. The ARM-PVMC system uses PVMC to access data from the *Main Memory*.

Three types of vector data structures (V1, V2, and V3) are selected to test the performance of system architecture. The V1 vector has fixed strides between two elements, and its memory accesses are predictable at compile time. The V2 memory accesses are not predictable at compile but known at run-time, the distance between two consecutive elements of V2 is greater than the cache line size. The V3 memory accesses are not known at run-time, we used a random address generator that provides address before computation.

Figure 13 shows the number of clock cycles taken *ARM-PVMC*, and *ARM-GMC* to access and processes V1, V2 and V3 vectors. Y-axis presents the number of clocks in logarithmic scale. The x-axis shows the applications with V1, V2 and V3 vector data access patterns. While perforating vector multiplication on V1 type vector, the *ARM-PVMC* achieves 4.12x of speedups against the *ARM-GMC*. The V1 addresses are aligned; therefore, the ARM system uses multiple direct memory access calls which require the start address, the end address and the size of the transfer. The *ARM-PVMC* organizes the known address in descriptors which reduces the run-time address generation and address management time. The ARM snoop control unit (SCU) reuses and prefetches the data in parallel with the computation, but there is still on-chip bus management and *Main Memory* data transfer issues. For vector V2 addition, the *ARM-PVMC* achieves 12.68x of speedup. The V2 has run-time predictable stride size, having a size greater than the cache line. The baseline system uses multiple transfer calls to access the vector, which generates address generation, on-chip bus arbitration and the *Main Memory* bank section time. The *ARM-GMC* uses multiple load/store and DMA transfer call to access V2 vector; this adds on-chip bus delay. The *ARM-PVMC* efficiently handles strides at run-time and manages address/data of access patterns, translates/recorders in hardware, in parallel with ARM processor cores. As the vector addresses are in the form of descriptors the PVMC on-chip bus and the *memory controller* manage data transfers in a single or few bursts. The *Specialized Memory* places complete data structure in continuous format and feed it to processing cores that reduce the local memory management time. The *Specialized Memory* banks read/write operations are performed in

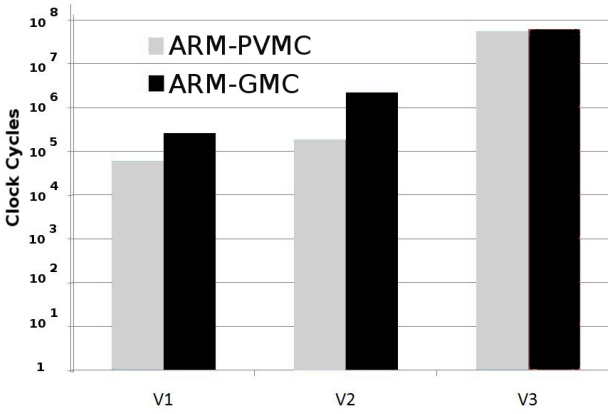

Fig. 13. ARM: Generic Memory Controller and PVMC based System

parallel. The vector addition of type V3 vectors gives 1.21x of speedup. The V3 vector requires pointer/irregular data transfer calls, which generates address management, bus arbitration, scheduling and the *Main Memory* delays. As the address are not known, the *ARM-PVMC Memory Manager* and *Memory Controller* do not able to work in parallel. The *ARM-GMC* uses generic *Local Memory*, *Bus Management* and the *Main Memory* units. This results in a settlement on the possible performance because of the generic units requires extra handshaking and synchronization time. In order to achieve performance *ARM-PVMC* bypasses the generic system units and introduces *Specialized Memory*, *Memory Management* and *Main Memory*. The *PVMC* internal units have the ability to operate independently, in parallel with each other and *ARM-PVMC* achieves maximum performance when all of its units works in parallel.

7 Related Work

Yu et al. propose VIPERS [21], a vector architecture that consists of a scalar core to manage data transfers, a vector core for processing data, an address generation logic, and a memory crossbar to control data movement. Chou et al. present the VEGAS [11] vector architecture with a scratchpad to read and write data and a crossbar network to shuffle vector operations. VENICE [22] is an updated version of VEGAS, with scratchpad and DMA that reduces data redundancy. VENICE has limitations regarding the rearrangement of complex data with scatter/gather support. Yiannacouras et al. propose the VESPA [10] processor that uses a configurable cache and hardware prefetching of a constant number of cache lines to improve the memory system performance. The VESPA system uses wide processor buses that match the system cache line sizes. VIPERS and VEGAS require a scalar Nios processor that transfers data between the scratchpad and the main memory. A crossbar network is used to align and arrange on-chip data. The PVMC eliminates the crossbar network and the limitation of using a scalar processor for data transfer. PVMC manages addresses in hardware with the pattern descriptors and accesses data from *Main Memory* without support of a scalar processor core. The PVMC data manager rearranges on-chip data using the *Buffer Memory* without a complex crossbar network which allows the vector processor to operate at higher clock rates.

McKee et al. [23] introduce a Stream Memory Controller (SMC) system that detects and combines streams together at program-time and at run-time prefetches read-streams, buffers write-streams, and reorders the accesses to use the maximum available memory bandwidth. The SMC system describes the policies that reorder streams with a fixed stride between consecutive elements. The PVMC system prefetches both regular and irregular streams and also supports dynamic streams whose addresses are dependent on run-time computation. McKee et al. also proposed the Impulse memory controller [24] [25], which supports application-specific optimizations through configurable physical address remapping. By remapping the physical addresses, applications can manage the data to be accessed and cached. The Impulse controller works under the command of the operating system and performs physical address remapping in software, which may not always be suitable for HPC applications using hardware accelerators. PVMC remaps and produces physical addresses in the hardware unit without the overhead of

operating system intervention. Based on its C/C++ language support, PVMC can be used with any operating system that supports the C/C++ stack.

A scratchpad is a low latency memory that is tightly coupled to the CPU [26]. Therefore, it is a popular choice for on-chip storage in real-time embedded systems. The allocation of code/data to the scratchpad memory is performed at compile time leading to predictable memory access latencies. Panda et al. [27] developed a complete allocation strategy for scratchpad memory to improve the average-case program performance. The strategy assumes that the access patterns are known at compile time. Suhendra et al. [28] aims at optimizing the worst-case performance of memory access tasks. However, in that study, scratch-pad allocation is static having static and predictable access patterns that do not change at run-time, raising performance issue when the amount of code/data is much larger than the scratchpad size. Dynamic data structure management using scratchpad techniques are more effective in general because they may keep the working set in scratchpad. This is done by copying objects at predetermined points in the program in response to execution [29]. Dynamic data structure management requires a dynamic scratchpad allocation algorithm to decide where copy operations should be carried out. A time-predictable dynamic scratchpad allocation algorithm has been described by Deverge and Puaut [29]. The program is divided into regions, each with a different set of objects loaded into the scratchpad. Each region supports only static data structures. This restriction ensures that every program instruction can be trivially linked to the variables it might use. Udayakumaran et al. [30] proposed a dynamic scratchpad allocation algorithm that supports dynamic data structures. It uses a form of data access shape analysis to determine which instructions can access which data structures, and thus ensures that accesses to any particular object type can only occur during the regions where that object type is loaded into the scratchpad. However, the technique is not time-predictable, because objects are spilled into external memory when insufficient scratchpad space is available. The PVMC *Address Manager* arranges unknown memory access at run-time in the form of pattern descriptors. The PVMC *Data Manager* rearranges on-chip data using the *Buffer Memory* without a complex crossbar network, which allows the vector processor to operate at higher clock rates.

A number of off-chip DMA Memory Controllers have been suggested in the past. The Xilinx XPS Channelized DMA Controller [31], Lattice Semiconductor's Scatter-Gather Direct Memory Access Controller IP [32] and Altera's Scatter-Gather DMA Controller [33] cores provide data transfers from non-contiguous blocks of memory by means of a series of smaller contiguous transfers. The data transfer of these controllers is regular and is managed/controlled by a microprocessor (Master core) using a bus protocol. PVMC extends this model by enabling the memory controller to access complex memory patterns.

Hussain et al. proposed a vector architecture called programmable vector memory controller (PVMC) [34] and its implementation on an Altera Stratix IV 230 FPGA device. The PVMC accesses memory patterns and feed them to soft vector processor architecture. The Advanced Programmable Vector Memory Controller supports application specific accelerator, scalar soft vector core processor and hard core ARM processor architectures. The advanced architecture is tested with the applications having complex complex memory patterns.

8 Conclusion

The memory unit can easily become a bottleneck for vector accelerators. In this paper, we have suggested a memory controller for vector processor architectures that manages memory accesses without the support of a scalar processor. Furthermore, to improve the on-chip data access a *Buffer Memory* and a *Data Manager* are integrated that efficiently access, reuse, align and feed data to the vector processor. A *Multi DRAM Access Unit* is used to improve the *Main Memory* bandwidth which manages the memory accesses of multiple *SDRAM*s. The experimental evaluation based on the VESPA vector system demonstrates that the PVMC based approach improves the utilization of hardware resources and efficiently accesses *Main Memory* data. The benchmarking results show that PVMC achieves between 2.01x to 4.53x of speedup for 10 applications, consumes 2.56 to 4.04 times less energy and transfers different data set patterns up to 1.40x and 2.12x faster than the baseline vector system. In the future, we plan to embed run-time memory access aware descriptors inside PVMC for vector-multicore architectures.

References

1. Visual computing technology from NVIDIA. http://www.nvidia.com/.
2. Roger Espasa, Mateo Valero, and James E Smith. Vector architectures: past, present and future. In *12th international conference on Supercomputing*, 1998.
3. Christos Kozyrakis and David Patterson. Overcoming the limitations of conventional vector processors. In *ACM SIGARCH Computer Architecture News, 2003*.
4. Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 129–140. ACM, 2011.
5. Tassadaq Hussain, Miquel Pericas, Nacho Navarro and Eduard Ayguade. Implementation of a Reverse Time Migration Kernel using the HCE High Level Synthesis Tool.
6. Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady and Mateo Valero. Advanced Pattern based Memory Controller for FPGA based HPC Applications. In *International Conference on High Performance Computing & Simulation*, page 8. ACM, IEEE, 2014.
7. Tassadaq Hussain, Miquel Pericas, Nacho Navarro and Eduard Ayguade. PPMC: Hardware Scheduling and Memory Management support for Multi Hardware Accelerators. In *FPL 2012*.
8. Embedded Development Kit EDK 10.1i. *MicroBlaze Processor Reference Guide*.
9. Nios II: Processor Reference Handbook, 2009.
10. Peter Yiannacouras, J Gregory Steffan, and Jonathan Rose. Vespa: portable, scalable, and flexible fpga-based vector processors. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70. ACM, 2008.
11. Christopher H Chou, Aaron Severance, Alex D Brant, Zhiduo Liu, Saurabh Sant, and Guy GF Lemieux. Vegas: soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 15–24. ACM, 2011.
12. Russell Richard M. The CRAY-1 computer system.
13. Cheng Hui. Vector pipelining, chaining, and speed on the IBM 3090 and cray X-MP.

14. Weiss Michael. Strip mining on SIMD architectures. In *Proceedings of the 5th international conference on Supercomputing*. ACM, 1991.
15. Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady, Mateo Valero and Amna Haider. Stand-alone Memory Controller for Graphics System. In *The 10th International Symposium on Applied Reconfigurable Computing (ARC 2014)*. ACM, 2014.
16. Tassadaq Hussain and Amna Haider. PGC: A Pattern-Based Graphics Controller. International Journal of Circuits and Architecture, 2014.
17. Tassadaq Hussain, Muhammad Shafiq, Miquel Pericas, Nacho Navarro and Eduard Ayguade. PPMC: A Programmable Pattern based Memory Controller. In *ARC 2012*.
18. Tassadaq Hussain, Miquel Pericas, Nacho Navarro and Eduard Ayguade. Reconfigurable Memory Controller with Programmable Pattern Support. Jan, 2011.
19. Peter Yiannacouras, Jonathan Rose, and J Gregory Steffan. The microarchitecture of FPGA-based soft processors. International conference on Compilers, architectures and synthesis for embedded systems 2005.
20. Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.
21. Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. Vector processing as a soft processor accelerator. volume 2, page 12. ACM, 2009.
22. Aaron Severance and Guy Lemieux. Venice: A compact vector processor for fpga applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 261–268. IEEE, 2012.
23. Sally A McKee, William A Wulf, James H Aylor, Robert H Klenke, Maximo H Salinas, Sung I Hong, and Dee AB Weikle. Dynamic access ordering for streamed computations. volume 49, pages 1255–1271. IEEE, 2000.
24. John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis,Chen-Chi Kuo, Ravindra Kuramkote,Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a Smarter Memory Controller. Prentice-Hall, Inc., January 1999.
25. Lixin Zhang, Zhen Fang, Mike Parker, Binu K Mathew, Lambert Schaelicke, John B Carter, Wilson C Hsieh, and Sally A McKee. The impulse memory controller. volume 50, pages 1117–1132. IEEE, 2001.
26. Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M Balakrishnan, and Peter Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *15th International Symposium on System Synthesis, 2002.*
27. Panda Preeti Ranjan, Dutt Nikil D and Nicolau Alexandru. *Memory issues in embedded systems-on-chip: optimizations and exploration*. Springer, 1999.
28. Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium, 2005. RTSS 2005.*
29. J-F Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *19th Euromicro Conference on Real-Time Systems, 2007. ECRTS'07.*
30. Udayakumaran Sumesh, Dominguez Angel and Barua Rajeev. Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems (TECS).
31. Xilinx. *Channelized Direct Memory Access and Scatter Gather*, February 25, 2010.
32. Lattice Semiconductor Corporation. *Scatter-Gather Direct Memory Access Controller IP Core Users Guide*, October 2010.
33. Altera Corporation. *Scatter-Gather DMA Controller Core, Quartus II 9.1*, November 2009.

34. Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady and Mateo Valero. PVMC: Programmable Vector Memory Controller. In *The 25th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE ASAP 2014 Conference, 2014.