PVMC: Programmable Vector Memory Controller

Tassadaq Hussain[1,2], Oscar Palomar[1,2], Osman Unsal[1], Adrian Cristal[1,2,3], Eduard Ayguadé[1,2], Mateo Valero[1,2]
[1] Computer Sciences, Barcelona Supercomputing Center, Barcelona, Spain
[2] Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain
[3] Artificial Intelligence Research Institute (IIIA), Centro Superior de Investigaciones Científicas (CSIC), Barcelona, Spain
Email: {first}.{last}@bsc.es

***Abstract*—In this work, we propose a Programmable Vector Memory Controller (PVMC), which boosts noncontiguous vector data accesses by integrating descriptors of memory patterns, a specialized local memory, a memory manager in hardware, and multiple DRAM controllers. We implemented and validated the proposed system on an Altera DE4 FPGA board. We compare the performance of our proposal with a vector system without PVMC as well as a scalar only system. When compared with a baseline vector system, the results show that the PVMC system transfers data sets up to 2.2x to 14.9x faster, achieves between 2.16x to 3.18x of speedup for 5 applications and consumes 2.56 to 4.04 times less energy.**

I. Introduction

Data Level Parallel (DLP) accelerators such as GPUs [1] and Vectors [2], [3], [4], are getting popular due to their high performance per area. 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 decode bandwidth and exploit DLP for data intensive applications i.e. matrix & media oriented, etc. While hard-core architectures offer excellent packaging and communication advantages, a soft vector core on FPGA offers the advantage of flexibility and lower part costs. A soft vector architecture is very efficient for HPC applications, because it can be scaled depending upon the required performance and available FPGA resources. Therefore a number of FPGA based soft vector processors have been proposed [5] [6]. A soft vector unit typically comprises a parameterized number of vector lanes, a vector register file, a vector memory unit and a crossbar network that shuffles vector operands.

Typically, the vector processor is attached to a 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 [7] 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.

In this paper we propose a 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 the 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 integrate the proposed system with an open source soft vector processor, VESPA [5] and used an Altera Stratix IV 230 FPGA device. We compare the performance of the 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 2.2x to 14.9x faster, achieves between 2.16x to 3.18x of speedup for 5 applications and consumes 2.56 to 4.04 times less energy.

II. Vector Processor

A vector processor is also known as a "single instruction, multiple data" (SIMD) CPU [8], 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(a). The number of vector lanes determines the number of ALUs and elements that can be processed in parallel. 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. The vector processor uses a scalar core for all control flow, branches, stack, heap, and input/output ports.

Modern vector memory units use local memories (cache or scratchpad) and transfer data between the main memory and the VLs. 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 memory crossbar (MC) is used to route each byte of the cache line (CL) accessed simultaneously to any lane. 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 research leading to these results has received funding from the European Research Council under the European Unions 7th FP (FP/2007-2013) / ERC GA n. 321253. It has been partially funded by the Spanish Government (TIN2012-34557).

© 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works

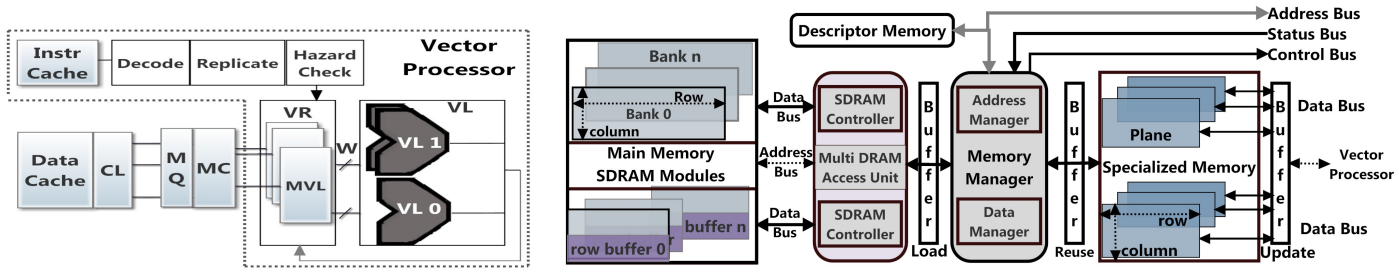


Fig. 1. (a) Generic Vector Processor (b) PVMC: Vector System

the number of lanes, this process will take several cycles. Vector chaining [9] 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 a loop is vectorized and the original loop count is larger than the MVL, a technique called strip-mining is applied [10]. The body of the strip-mined vectorized loop operates on blocks of MVL elements.

III. Programmable Vector Memory Controller

The Programmable Vector Memory Controller (PVMC) architecture is shown on Figure 1(b), 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 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 SDRAM controllers.

A. Bus System

As the number of processing cores and the capacity of memory components increase the system requires a high speed bus interconnection network that connects the processor cores and memory modules [11]. 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 *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.

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

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

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}, $\text{Controller}_{clock}$ and $\text{SDRAM}_{bus\ width}$ represent the number of separate *SDRAM module*s, the clock speed of each SDRAM controller and the data width of SDRAM 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 III-B3). 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. A 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.

B. Memory Hierarchy

The PVMC memory hierarchy consists of descriptor memory, specialized memory, register memory and main memory.

1) Descriptor Memory: The descriptor memory [12],[13] 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 next. The main memory and local memory address parameters specify the memory locations to read and write data respectively. 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.

2) Buffer Memory: The *buffer memory* architecture implements the following features:

- Data realignment to match vector lanes. It aligns data when input and output vector elements are not the same.
- 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.
- 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 the *Specialized Memory* data (see III-B3). 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) 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. Double-buffering can be used to overlap almost completely computation and memory transfer for each read and write specialized memory. In case of double-buffering, the PVMC prefetches data from main memory into the specialized memory without interrupting the datapath of the vector lanes. In the meantime vector lanes keep working on data from the specialized memory that has already been accessed.

The specialized memory structure is divided into multiple planes as shown in Figure 1(b), 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. PVMC supports large vector, 2D and 3D tiled specialized memory structures.

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 internally multiple independent banks that can operate in parallel. Each bank represents multiple arrays (rows and column) and it can be accessed in parallel with other banks.

C. 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 IV-D). 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.

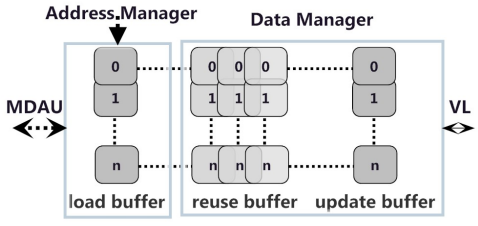


Fig. 2. Data Memory Buffers: Load, Reuse & Update

1) Address Manager: The Address Manager takes a vector transfer instruction and reads the appropriate descriptor memory. Depending on the access pattern the address manager uses single or multiple descriptors, maps and rearranges addresses in hardware. The Address Manager saves mapped addresses into its address buffer for further reuse.

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 2) to load, rearrange and write vector data. When input and output vectors are not aligned the data manager shuffles data between lanes. In case of strip mining 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 III-D) 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.

D. 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 busses. 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 *SDRAM Controller*s using separate data buses, which increases the memory bandwidth. There is one *SDRAM Controller* per *SDRAM module*. Each *SDRAM 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*.

IV. PVMC Functionality

In this section we discuss the important challenges faced by the memory unit of soft vector processors and explain our solution.

A. 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 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 cache resulting in considerable waste of resources. PVMC manages non-unit stride memory accesses similar to unit-stride. 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 it for future accesses.

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 aligns the vector data. 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 input and output vector elements are not the same. It also reuses and updates existing vector data and loads data which is not present in the *specialized memory*.

B. 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 an 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.

C. 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 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 requires extra time to initialize addresses, synchronise on-chip buses and SDRAMs. The PVMC MDAU uses a single descriptor for unit and non-unit stride accesses which improves the memory bandwidth by transferring descriptors to the memory controllers, rather than individual references and by accessing data from multi-SDRAM devices.

D. Programming Vector Accesses

Figures 3 (a) and (b) show vector loops (with MVL of 64) for a 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 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 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 3(c), 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 is available for processing. In this way PVMC pipelines the data transfers

```
for(i=0; i<length; i+=64)
{
VLD.S (/*Main Memory(MM)*/ 0x00000000+i,
/*Vector Register (VR)*/, VR0,
/*Stride (ST)*/ 0x04);
VLD.S (/*MM*/ 0x00000100+i, /*VR*/, VR1,
/*ST*/ 0x04);
VADD VR0, VR1, VR2

VLD.S (/*MM*/ 0x00000200+i, /*VR*/, VR3,
/*ST*/ 0x04);
VADD VR2, VR3,

VST.S (/*MM*/ 0x10000000+i,
/*VR*/, VR3, /*ST*/ 0x04);
}
```

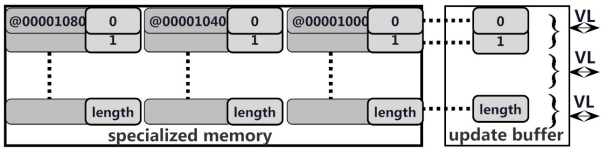
(a)

```
PVMC_VLD (/*MM*/ 0x00000000, /*Local Memory(LM)*/
0x00001000, /*Size (SZ)*/ length, /*ST*/ 0x04 );
PVMC_VLD (0x00000100, 0x00001040, length, 0x04);
PVMC_VLD (0x00000200, 0x00001080, length, 0x04);

for(i=0; i<length; i+=64)
{   VLD(/*LM*/ 0x00001000+i,/*VR*/, VR0);
    VLD(/*LM*/ 0x00001040+i,/*VR*/, VR1);
VADD VR0, VR1, VR2
    VLD( /*LM*/ 0x00001080+i,/*VR*/, VR3);
VADD VR2, VR3, VR3
    VST( /*LM*/ 0x11000000+i,/*VR*/, VR3);     }

PVMC_VST (/*MM*/ 0x10000000, /*LM*/ 0x11000000,
          /*SZ*/ length, /*ST*/ 0x04 );
```

(b)



(c)

Fig. 3. (a) Vector Loop (b) PVMC Vector Loop (c) PVMC Data Transfer Example

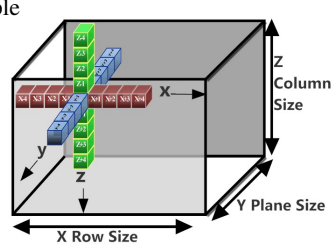


Fig. 4. 3D Stencil Vector Access

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 in an API. The memory access information is included in the PVMC header file and provides function calls (e.g. 3D_STEN(), 3D_TILE(), 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. 3D-Stencil, 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 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 4. By combining these descriptors, the PVMC exchanges 3D data between the main memory and the *specialized memory* buffer. The value X, Y and Z define the width (row_size), height (column_size) and length (plane_size) respectively of the 3D memory block. The 3D-Stencil has x, z and y vectors having direction of row, column and plane 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* × *column_size*.

V. 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*.

A. The VESPA System

The FPGA based vector system is shown in Figure 5(a). The system architecture is further divided into the *Scalar core*, the *Vector core* and *Memory System*.

1) Scalar Core: A SPREE [14] 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.

2) Vector Core: A soft vector processor called VESPA (Vector Extended Soft Processor Architecture) [5] 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 maximum vector length (MVL) of 128.

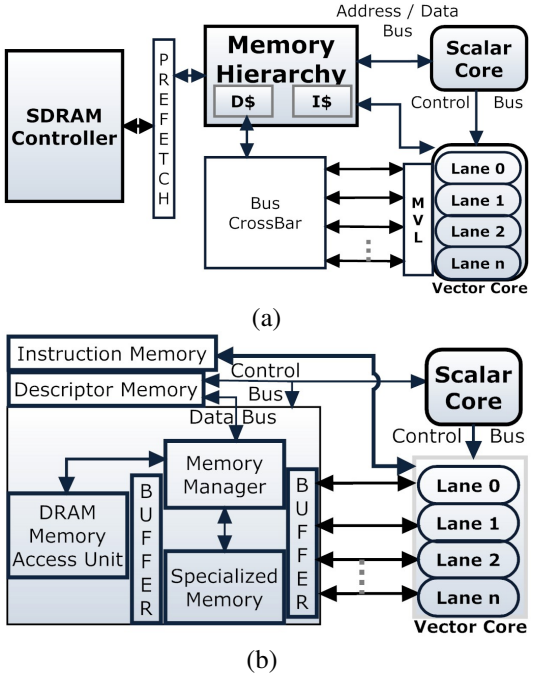


Fig. 5. (a) Baseline VESPA System (b) PVMC System

3) Memory System: The baseline VESPA vector memory unit (shown in Figure 5(a)) includes a 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 which 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.

B. The Proposed PVMC System

The PVMC based vector system is described in Sections II and III and shown in Figure 5(b). The major difference between the PVMC and VESPA systems is the memory system. 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.

C. The Baseline Nios System

The Nios II processor scalar [15] is a 32-bit embedded-processor architecture designed specifically for the Altera family of FPGAs. The Nios II architecture is a 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 (SD-DMA) along with SDRAM controller is used that handles multiple data transfers efficiently.

D. Applications

Table I shows the application kernels which are executed on the vector systems along with their memory access patterns. The set of applications cover 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.

VI. 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.

TABLE I. Brief description of application kernels

Application	FIR	1D Filter	Tri-Diagonal	Matrix Multiplication	Gaussian	Motion Estimation	3D-Stencil
Access Pattern	Stream	1D Block	Diagonal	Row & Column	2D Block	2D Block	3D-Tiling

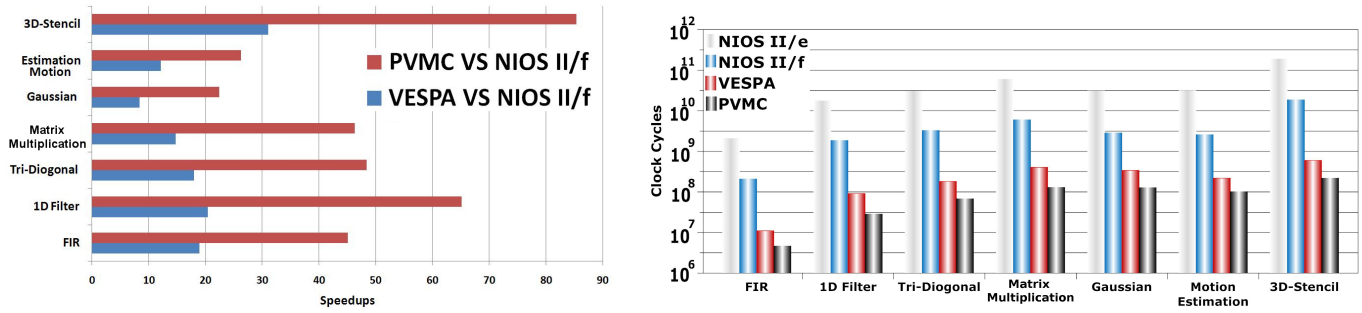


Fig. 6. (a) Speedup of: PVMC and VESPA over Nios II/f (b) Vector & Scalar Systems: Application Kernels Execution Clocks

A. 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 II (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 by increasing the number of vector lanes from 1 to 64 requires larger crossbar multiplexer switches and operates at lower frequency. The PVMC specialized memory uses separate read and write specialized memories which reduces 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 II (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 III 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 and write specialized memories, therefore it occupies twice the number of BRAM bits. The data manager of the PVMC memory system occupies 3 to 5 times less resources than the VESPA memory system. Column *Main Memory* presents the resource utilization of the SDRAM controllers. The VESPA system does not support dual 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.

B. Performance Comparison

For performance comparisons, we use the application of Table I. 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.

Figure 6)(a) shows the speedups of VESPA and PVMC systems over Nios II/f. Results show that vector execution with the PVMC is 8.3x and 31.04x faster than the Nios II/f. Results for Nios II/e are not shown in Figure 6)(a) due to lack of space. 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 speed ups over the scalar processor NIOS are caused by using SPREE as the scalar unit of the vector processor, we execute FIR, Matrix Multiplication and 3D-Stencil application kernels on a SPREE scalar processor, i.e. with the vector processor disabled. While comparing performance of FIR, Matrix Multiplication 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 SPREE scalar processor. The results show that Nios II/f performs better than Nios II/e and SPREE scalar processors.

By using the PVMC system, the results show (Figure 6(b)) that the FIR kernel achieves 2.37x of speedup over VESPA.

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 II. (A) LOCAL BUS MAXIMUM FREQUENCY (MHZ) (B) GLOBAL BUS MAXIMUM FREQUENCY (MHZ)

TABLE III. 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

The application kernel has streaming data accesses and requires a single descriptor to access a stream which 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 Matrix Multiplication 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. 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 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.

C. Dynamic Power & Energy

To measure voltage and current the DE4 board provides a resistor to sense current/voltage and 8-channel differential 24-bit analogue to digital convertors. Table IV 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 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.

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 II/f @100		9744 , 10126	2.086	1.686	3.760	82.56
Nios II/f @200		12272 , 10256	3.109	2.513	5.822	48.379
VESPA @100	1	7227 , 7878	1.54	2.24	3.78	101.99
	4	7867 , 12193	1.874	2.24	4.17	60.04
	16	10090 , 31081	3.191	2.24	5.353	14.36
	32	13273 , 57878	4.666	2.24	6.29	9.42
	64	19641 , 103857	5.57	2.25	7.78	7.026
PVMC @100	1	5227 , 5587	1.1	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 , 88,934	4.01	2.11	6.13	1.75

TABLE IV. SYSTEMS: RESOURCE, POWER AND ENERGY UTILIZATION

D. Bandwidth

In this section, we measure the bandwidth of the PVMC, VESPA and Nios ll/f systems by reading and writing complex 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 processing cores have 32 bit on-chip data bus operating at 100 MHZ that provides a maximum bandwidth of 400 MB. The PVMC can achieve maximum bandwidth by using data transfer size equal to the data set. In order to check the effects of memory, bus and address management units over the system bandwidth, we transfer data between processor and memory using different pattern and transfer sizes. The X-axis presents random load/store, streaming, a 2D and 3D tiled data sets of 2MB that are read and written from/to the main memory. The load/store access patterns read/write 4B from a random location. A single streaming access pattern accesses 1KB of stream and a 2D access pattern reads/writes a 2D block with row and column size of 1KB. The 3D tiled benchmark reads a 3D tile of 128x128x128 bytes (rows, column and plane) and writes it back to the main memory. The Y-axis shows the bandwidth in MB per second for single and double *SDRAM Controller*(s). Figure 7 shows a bar chart of different data transfers for the PVMC, VESPA and Nios ll/f -based systems. While using single and double *SDRAM Controller*(s), the results show that PVMC random load/store data transfers are 2.2x, 3.4x and 3.5x, 2.3x times faster than VESPA and Nios ll/f systems respectively. The load/store data transfers require large control information (e.g. bus and memory initialization, etc.) that limits the bandwidth. The data transfer is further improved up to 4.9x, 9.95x and 4.5x, 4.8x times while transferring streaming data. While transferring 2D tiled data, the PVMC archives 8.8x, 14.9x and 8.2x, 8.94x of speedup. For complex data transfers, PVMC improves bandwidth 10.3, 9.8 and 16.4, 12 times. The VESPA system uses a single data bus to transfer to/from main memory, therefore it is unable to get the benefit from double *SDRAM Controller*s. The Nios ll/f system uses a SG-DMA controller that transfers data using unit-stride and forces to follow bus protocol. The PVMC system has a dedicated data bus for each *SDRAM controller* therefore, it efficiently accesses data from single- or multi- main memories and manages multi-*SDRAM Controller*(s) without support of a microprocessor. The PVMC complex patterns use few descriptors that reduce run-time address generation and address request/grant delay. For 3D

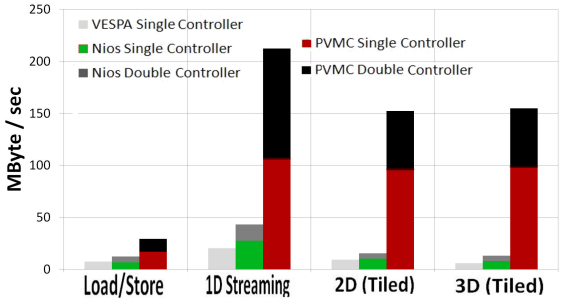


Fig. 7. Vector & Scalar Systems: Memory Bandwidth

tiled data transfer PVMC improves bandwidth by managing addresses at compile-time and by accessing data from multi-DRAM devices and multiple banks in parallel.

VII. Related Work

Yu et al. propose VIPERS [16], 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 [6] vector architecture with a scratchpad to read and write data and a crossbar network to shuffle vector operations. VENICE [17] is an updated version of VEGAS, with scratchpad and DMA that reduces data redundancy. VENICE has limitations of rearranging complex data with scatter/gather support. Yiannacouras et al. propose the VESPA [5] 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 matches 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, that allows the vector processor to operate at higher clock rates.

McKee et al. [18] 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 [19] [20], 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 number of off-chip DMA Memory Controllers have been suggested in the past. The Xilinx XPS Channelized DMA Controller [21], Lattice Semiconductor's Scatter-Gather Direct Memory Access Controller IP [22] and Altera's Scatter-Gather DMA Controller [23] 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/controled by a microprocessor (Master core) using a bus protocol. PVMC extends this model by enabling the memory controller to access complex memory patterns.

VIII. Conclusion

The memory unit can easily become the 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 specialized 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 2.16x to 3.18x of speedup for 5 applications, consumes 2.56 to 4.04 times less energy and transfers different data set patterns up to 2.2x and 14.9x 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 et al. Vector architectures: past, present and future. In *12th international conference on Supercomputing*, 1998.

[3] C. Kozyrakis et al. Overcoming the limitations of conventional vector processors. In *ACM SIGARCH Computer Architecture News, 2003*.

[4] Yunsup Lee et al. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. 2011.

[5] Peter Yiannacouras et al. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *The CASES 08*.

[6] Christopher H Chou et al. Vegas: soft vector processor with scratchpad memory. In *Proceedings of the international symposium on FPGA 2011*.

[7] Tassadaq Hussain et al. Implementation of a reverse time migration kernel using the hce high level synthesis tool. *International Conference on Field-Programmable Technology 2011*.

[8] Richard M Russell. The cray-1 computer system. *Communications of the ACM 1978*.

[9] Hui Cheng. Vector pipelining, chaining, and speed on the ibm 3090 and cray x-mp. *IEEE, Computer 1999*.

[10] Michael Weiss. Strip mining on simd architectures. In *Proceedings of the 5th international conference on Supercomputing*. ACM, 1991.

[11] Tassadaq Hussain et al. Stand-alone memory controller for graphics system. In *The 10th International Symposium on Applied Reconfigurable Computing (ARC 2014)*.

[12] Tassadaq Hussain et al. PPMC: A Programmable Pattern based Memory Controller. In *8th International Symposium on ARC 2012*.

[13] Tassadaq Hussain et al. Recongurable memory controller with programmable pattern support. *5th HiPEAC Workshop on Reconfigurable Computing (WRC) 2007*.

[14] Peter Yiannacouras et al. The microarchitecture of fpga-based soft processors. In *International conference on Compilers, architectures and synthesis for embedded systems 2005*.

[15] Nios ii: Processor reference handbook, 2009.

[16] Jason Yu et al. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems, 2009*.

[17] Aaron Severance et al. Venice: A compact vector processor for fpga applications. In *International Conference on Field-Programmable Technology 2012*.

[18] Sally A. McKee et al. Dynamic access ordering for streamed computations. *IEEE Trans. Computer. November 2000*.

[19] John Carter et al. Impulse: Building a Smarter Memory Controller. *5th International Symposium on HPCA*, January 1999.

[20] Lixin et al. Zhang. The impulse memory controller. *Computers, IEEE Transactions on*, 2001.

[21] Xilinx. *Channelized Direct Memory Access and Scatter Gather*, February 25, 2010.

[22] Lattice Semiconductor Corporation. *Scatter-Gather Direct Memory Access Controller IP Core Users Guide*, October 2010.

[23] Altera Corporation. *Scatter-Gather DMA Controller Core, Quartus II 9.1*, November 2009.