EMVS: Embedded Multi Vector-core System

Tassadaq Hussain[1,3], Amna Haider[1,3], Adrian Cristal[2,4,5] and Eduard Ayguadé[2]
[1] Riphah International University Islamabad, Pakistan
[2] Barcelona Supercomputing Center, Spain
[3] UCERD Islamabad, Pakistan
[4] BarcelonaTech, Spain
[5] IIIA Artificial Intelligence Research Institute CSIC Spanish National Research Council
tassadaq@ucerd.com

Abstract

With the increase in the density and performance of digital electronics, the demand for a power-efficient high-performance computing (HPC) system has been increased for embedded applications. The existing embedded HPC systems suffer from issues like programmability, scalability, and portability. Therefore, a parameterizable and programmable high-performance processor system architecture is required to execute the embedded HPC applications. In this work, we proposed an embedded multi vector-core system (EMVS) which executes the embedded application by managing the multiple vectorized tasks and their memory operations. The system is designed and ported on an Altera DE4 FPGA development board. The performance of EMVS is compared with the Heterogeneous Multi-Processing Odroid XU3, Parallela and GPU Jetson TK1 embedded systems. In contrast to the embedded systems, the results show that EMVS improves 19.28 and 10.22 times of the application and system performance respectively and consumes 10.6 times less energy.

1. Introduction

With the improvement in the number of processor on a chip, it directs system architects to develop and use a processor architecture that exploits maximum parallelism and improve the performance of Embedded HPC applications. Custom digital system architectures give the best results for a specific application,

© 2018 Elsevier . This manuscript version is made available under the CC-BY-NC-ND 4.0 license
http://creativecommons.org/licenses/by-nc-nd/4.0/

but growing mask and development expenses restrict application specific processors. Even then, application specific hardware is inappropriate for the product that needs instantly change during run-time requirements. Programmable application specific processor [1] is the more flexible option, and they have developed to utilize particular kinds of parallelism familiar to certain types of embedded applications. The resulting devices have application-specific instruction sets with several parallel execution and synchronization models, making them difficult to program. Data Level Parallel (DLP) [2] accelerators give the maximum performance by processing multi data elements with a single instruction. Because of the fixed architecture, these hardware accelerators do not exploit DLP for a set of data-intensive applications.

Because of the increase and improvement in the digital electronics, many manufacturers [3] [4] provide high performance single board computer systems for a large range of embedded applications. Similarly, the increase in the performance and density of Field Programmable Gate Arrays (FPGA)s [5] [6] allows the embedded applications industry to develop low power, low cost and high-performance systems, which can execute complex real-time applications on a single chip. The latest FPGA [7] devices provide the programmable high performance reconfigurable hardware with the programmable system integration capabilities, including both the high bandwidth serial I/O and signal processing bandwidth, as well as the highest on-chip memory density. As the FPGA architectures become larger, it allows many low power, low frequency, and high-performance processing cores in the design. The FPGA systems direct system architects to develop and use a processor architecture that exploits maximum parallelism and improve the performance of Embedded HPC applications. However, these system architectures suffer from poor efficacy due to programmability, scalability and portability issues.

To deal with programmability, scalability and portability issues, in this work we propose an Embedded Multi Vector-core System (EMVS). Some salient features of proposed EMVS architecture are:

- Consists scalable vector processors (VP) with parameterizable vector lanes (VL).
- Uses on chip specialized vector memory and on-chip bus network that efficiently handles complex/irregular data patterns.
- Manages memory access patterns of multiple vector processors in hardware thus improves the system performance by prefetching these complex access

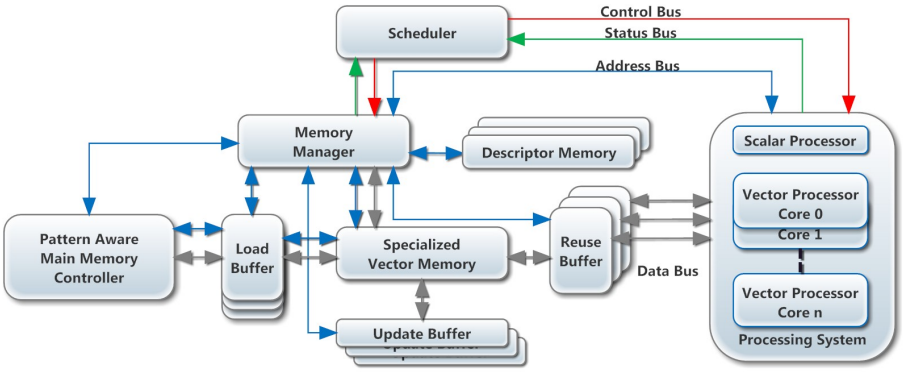

Figure 1: Embedded Multi Vector-core System Architecture

patterns in parallel with the vector processing.

- Uses an efficient run-time resource-aware *Scheduler* that manages and schedules multiple vector processor data and application tasks by using compile-time program and run-time automated scheduling policies.

- When compared the performance of EMVS with the Odroid XU3, Parallela, GPU Jetson TK1 systems, the results show that EMVS improves between 1.46 to 19.28 and 3.01 to 10.22 times application and system performance respectively and consumes 10.6 times less energy.

The rest of the work is arranged as follows. In Sections 2 and 3, we describe the architecture and working principle of EMVS respectively. The Section 4 gives details on the EMVS and hardware integration of multi-vector to a Baseline System. The performance and power comparison of EMVS by executing application kernels are provided in Section 5. Major differences between our proposal and state of the art are described in Section 6. Section 7 summarizes our main conclusions.

2. Embedded Multi Vector-core Processor System

The Embedded Multi Vector-core Processor System (EMVS) is shown in Figure 1. EMVS is divided into the *Processing System*, *Bus System*, the *Memory Hierarchy*, the *Scheduler*, the *Memory Manager* and the *Pattern Aware Main Memory Controller*.

2.1. Processing System

The EMVS *Processing System* uses a single Reduced Instruction Set Computer (RISC) processor and *Multi-Vector cores*. The RISC core is used to program the *Multi-Vector cores* and performs the scalar operations. The scalar core keeps working in parallel with the *Multi-Vector cores* except control instructions and scalar load/store instructions.

The *Multi-Vector cores* holds multiple Vector Processors (VP)s. A VP is also called as a *single instruction, multiple data* (SIMD) processor [8], that can work on an array of data in a pipelined mode, one element at a time applying a single instruction. For high performance, a VP can use multiple vector lanes (VL) to operate in lock-step on several elements of a vector in parallel. The structure of a single VP is given in Figure 2. The number of VLs defines the number of ALUs and elements that can be executed in parallel. The width (W) of any VP lane is 32-bit and can be set to 16-bit or 8-bit wide. The maximum vector length (MVL) limits the size of the vector register files (RF). Improving the MVL allows a single vector instruction to encapsulate extra parallel processes. This also increases the size of vector register file. Higher MVL values allow the application to utilize higher parallel processing in fewer vector instruction.

Depending upon the application requirement and hardware resources, the EMVS can integrate multiple VP cores in the architecture, which are handled by the scheduler (discussed in Section 2.5). In current prototype architecture on Altera DE4 FPGA, the EMVS can integrate a maximum of 8 VP cores each VP is capable of using the maximum of 128 VL. Each VP has its own program memory. At runtime a VP core reads the instructions from the program memory, decodes the instruction and proceeds to the replicate pipeline stage in parallel. The replicate

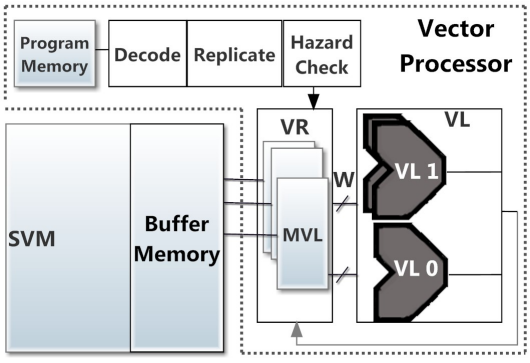

Figure 2: Single Vector Processor Architecture

pipeline stage divides the elements of work, requested by the vector instruction into smaller groups that are mapped onto the VLs. 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 VLs.

2.2. Bus System

The *Bus System* transfers control information, address, and data between scalar core, VP cores, and memory components. The *Status Bus* controls the multiple VP cores by utilizing the data transfer requests, acknowledgment, wait/ready and error/ok signals. The *Control Bus* manages the processing tasks, data transfer requests, and input/output operations. The *Address Bus* is used to classify the operations to read or write data from memory components or the *Processing System*. The *Address Bus* of EMVS is decoded and arbitrated by the *Memory Manager* (see Section 2.4). The *Data Bus* is used to transfer data between the *Main Memory* and the *Processing System*.

2.3. Memory Hierarchy

The EMVS *Memory Hierarchy* connects each byte in the *Local Memory* or the *Main Memory* to the VL of multiple VP cores. The *Memory Hierarchy* allocates and manages the *Local Memory* and the *Main Memory* address ranges for each VP cores. The EMVS *Memory Hierarchy* uses four types of memories which are the *Descriptor Memory*, the *Buffer Memory*, the *Specialized Vector Memory*, and the *Main Memory*. The *Descriptor Memory* [9] [10] hold the processing, data transfer, and scheduling information. The *Specialized Vector Memory* is used as the *Local Memory* that stores data set for processing by a VP core. Each VP core has separate *Specialized Vector Memory*. The *Main Memory* is shared between all the VP cores. Depending upon the vector data transfer the EMVS *Address Manager* (see Section 2.4) takes a single or multiple instructions from the *Descriptor Memory* and transfers a vector data set to/from the *Specialized Vector Memory* and the *Main Memory*. A single *Descriptor* holds the following parameters: *Processing Information*, *Processor ID*, *Main Memory* address, *Priority, Stream, Stride* and *Offset*. The *Processing Information* holds the vector or scalar processor arithmetic instruction. The *Processor ID* parameter specifies a processor core to execute the processing instruction. The *Main Memory* address specify the memory locations to read and write data. The *Priority* applies the order in which the processing

instructions are executed. *Stream* defines the number of data elements to be transferred. *Stride* specifies the jumps between two continuous memory addresses of a stream. The *Offset* parameter field is used for scatter/gather operations. It points the next scattered data transfer through the main address.

The *Buffer Memory* holds 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 Vector Memory* data. The *Specialized Vector Memory* has configurable structure and is divided into multiple planes [11], where each plane represents the rows and the columns. The *Specialized Vector Memory* uses the *Task ID* as the based address and has an address space separated from the *Main Memory*. A VP core holds a read *Specialized Vector Memory* and a write *Specialized Vector Memory*. The data of the read *Specialized Vector Memory* is sent directly to the vector lanes using the *Update Buffer*, and the results are written back into the write *Specialized Vector Memory*. The *Main Memory* is composed of structure of DRAMs and can be accessed using Pattern Aware Main Memory Controller (discussed in Section 2.6)

2.4. Memory Manager

The EMVS uses memory instructions for describing consecutive, strided, and *Offset* data transfer for the *Processing System* cores. The *Offset* transfer is used to move data in scatter/gather patterns. The *Memory Manager* of the EMVS processes and loads the requested address in the Memory Queue (MQ) for respective lane and then assign the data to the lanes. The EMVS *Memory Manager* uses *Address Manager* and *Data Manager*. At run-time, the *Memory Manager* takes memory address requests from the control bus using the *Scheduler* (shown Section 2.5) and transfers it to the *Address Manager*. The *Address Manager* uses a vector transfer instruction and reads the appropriate *Descriptor Memory* with data transfer information. It uses a single or multiple descriptors, maps addresses in hardware and saves assigned addresses into its address buffer for further reuse and rearranges them. The *Data Manager* performs on-chip data alignment and reuse. The *Data Manager* is used to rearrange the output data of vector lanes for reuse or update. The *Data Manager* uses the *Buffer Memory* to load, rearrange and write vector data. The *Data Manager* checks data requests from the *Specialized Vector Memory*, if data is available there then the data manager transfers it to the *Update Buffer*. If the data is not available then, the Memory Manager assigns the data information to the *Pattern Aware Main Memory Controller* (see 2.6) that accesses data from the *Main Memory* to the *Load Buffer*. The *Load Buffer* with the *Reuse*

Buffer make data alignment and reuse where needed, and load the *Update Buffer*. Next, the *Update Buffer* assigns data to the vector lanes.

2.5. Scheduler

The EMVS *Scheduler* manages multiple VP cores data transfer and processing tasks. The EMVS uses *Descriptor Memory* to calls data transfer patterns of each VP core, which reduces the input/output interfaces and multi-vector cores communication time. At run-time, the *Scheduler* receives multiple VP cores processing and data transfer tasks and perform data transfer and computation task partitioning, communication, and mapping of tasks on hardware. As generic Multi-core System scheduling, the *Scheduler* applies the *Symmetric* and *Asymmetric* scheduling policies [12]. The *Scheduler* also performs run-time adaptive scheduling [13]. The run-time adaptive scheduling is depending upon the VP core data transfer and processing tasks and free hardware devices.

The EMVS *Scheduler* takes the complete control of the multiple VP cores and executes the instructions by applying a lock and unlock instruction. The *Scheduler* executes a task/thread on a separate cores until it completes. The *Scheduler* generates an SIMD instruction on a specific VP clock cycle based on the arbitration policy. The EMVS *Scheduler* controls the run-time requests and program priorities of VP cores. Each VP core′s *task* includes a data transfer and processing operation. The EMVS programming model performs task partitioning at program time, and assign a priority state to the *Task ID*. The VP *task*s are categorized into three states, *busy* (VP core is processing on local buffer), *requesting* (VP core is idle), and *request & busy*.

The *Address Manager* of EMVS has particular *Descriptor Memory* (register set) for each VP core. These descriptors are masked with a request and interrupt signals. Once a VP task generates a request, the *Address Manager* starts memory operation for the VP core using its descriptors. After completion of data transfer operation and processing, the EMVS scheduler receives an interrupt (*ack*) signal from the memory management unit. This signal informs the scheduler to select the next data transfer request of the VP core to process. The scheduler captures the *ack* signal from the memory manager and assigns the *grant* signal to the appropriate VP core.

2.6. Pattern Aware Main Memory Controller

The *Pattern Aware Main Memory Controller (PAMMC)* reads/writes data from/to multiple *Main Memory* using several DRAM Controllers. *PAMMC* 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 translation from physical address to DRAM address and reads/writes data from/to its *SDRAM Module*.

3. EMVS Programming and Functionality

In this section, we discuss the important challenges faced by the embedded multi vector-core system and explain our solution. The section is further divided into two subsections: *Memory Hierarchy*, and *Programming EMVS*.

3.1. Memory Hierarchy

A generic VP system employs the cache hierarchy to increase the data locality by providing and reusing the needed data set to the processing cores. With a large number of vector processor core having multiple vector lanes and with strider data, the vector memory hierarchy does not meet the data spatial locality. EMVS improves the data spatial locality by accessing complex data elements into its *Specialized Vector Memory* and assigning them to VL using the *Buffer memory*. Non-unit stride accesses do not exploit spatial locality granted by caches which result in a significant loss of resources. EMVS manages non-unit stride memory accesses alike to unit-stride ones. Similar to a cache of VP, the EMVS *Specialized Vector Memory* tentatively holds data to speed up remaining accesses. Unlike a cache, data is purposely located in the *Specialized Vector Memory* at a known location, rather than automatically cached according to a set hardware strategy. The EMVS *Memory Manager* along with the *Buffer Memory* grip knowledge of unit and non-unit stridden transfer, update and reuse them for later accesses.

The EMVS *Memory Manager* performs vector chaining [14] that transfers the output of a VP instruction to a dependent vector instruction, avoiding the vector register file, therefore bypassing serialization and maintain multiple dependent vector instructions to perform together. Vector chaining can be coupled by increasing the amount of VPs and VLs. It requires free processing cores with a high MVL and increases the impact on the performance of vector chaining. When the loop is vectorized, and the primary loop number is higher than the MVL, the *Memory Manager* involves strip-mining technique [15]. The body of the strip-mined vectorized loop operates on blocks of MVL elements. EMVS *Memory Manager*

performs strip mining that transforms a loop into two nested loops: an outer strip-control loop with a step size of a multiple of the original loops step size, and an inner loop contains the original step size and the loop body. The EMVS *Memory Manager* performs strip mining [15] 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.

3.1.1. Memory Crossbars

To load and store data in a generic multi-core vector system, the vector register file is attached to the data cache by distributing read and write crossbars. If the input to the vector lanes is mismatched the vector processor requires an additional instruction that transfers and aligns the vector data. The crossbars rearrange bytes/half-words/words of their byte-offset against memory into word size at a distinct byte-offset in the vector register file. The size of the crossbars is restrained on one end by the overall width of the vector register file, and on the opposite side by the overall width of the on-chip memory/cache. The size and complexity of crossbars increase when the VP core is configured to operate more lanes. The EMVS uses the *Buffer Memory* to assign data to the vector register file which is simpler than using crossbar and data alignment. The *Buffer Memory* adjusts data when output vector elements are required to process with new input elements. It also reuses and updates existing vector data and loads data which is not present in the *Specialized Vector Memory*.

3.1.2. Address Registers

The vector processor applies address registers to access the *Main Memory* data. The memory unit utilizes address registers to process the valid address of a processing core in the *Main Memory*. A generic vector processor maintains unit-stride, stridden and offset transfers. The EMVS uses a separate register file to operate the *Descriptor Memory* using data transfer instructions to operate with the MIPS Instruction Set Architecture (ISA). The EMVS *Descriptor Memory* accesses larger than the MVL without changing the ISA. EMVS uses a single or multi descriptors to transfer various complex non-stride transfers.

```
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)

```
EMVS_VLD (/*Main Memory*/ 0x00000000, /*Local Memory */ 0x00001000, /*Size*/ length,
      /*Stride*/ 0x40 );
EMVS_VLD (/*Main Memory*/ 0x00000100, /*Local Memory */ 0x00001040, /*Size*/ length,
      /*Stride*/ 0x04 );
EMVS_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);
        }
EMVS_VST (/*Main Memory*/ 0x10000000, /*Local Memory */ 0x11000000, /*Size*/ length,
      /*Stride*/ 0x04 );
```

(c)

Figure 3: (a) Scalar Loop (b) Conventional Vector Loop (c) EMVS Vector Loop

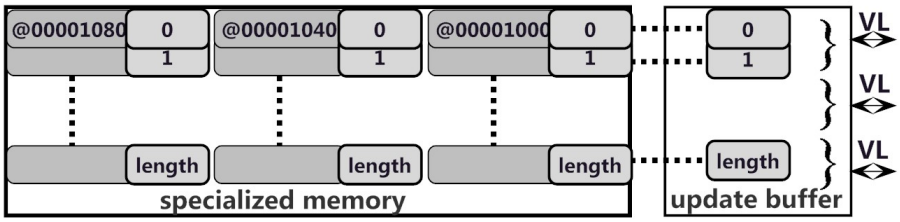

Figure 4: EMVS Data Transfer Example

3.1.3. Main Memory System

The generic *Main Memory* System employs a direct memory access (DMA) or Load/Store system to transfer data between the *Main Memory* and cache memory. Thus, a generic vector memory system uses a single DMA request to transfer a unit-stride access between the *Main Memory* and a cache line. But for complex or non-unit stridden accesses, the memory unit uses multiple DMA or Load/Store requests which need additional time to initialize addresses, synchronize on-chip buses and main memory banks. The EMVS PAMMC utilizes descriptors for a unit and non-unit stride transfer that grow the memory bandwidth by giving descriptors to the memory controllers, rather than using the individual addresses to access data from the multi-SDRAM devices.

3.2. Programming EMVS

A common concern, when using a multi VP core is the compiler support. A vector processor typically requires in-line assembly code that interprets vector instructions with an updated GNU assembler. In order to explain how EMVS is

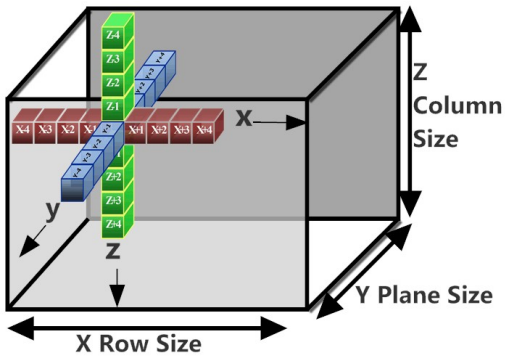

Figure 5: 3D Stencil Access

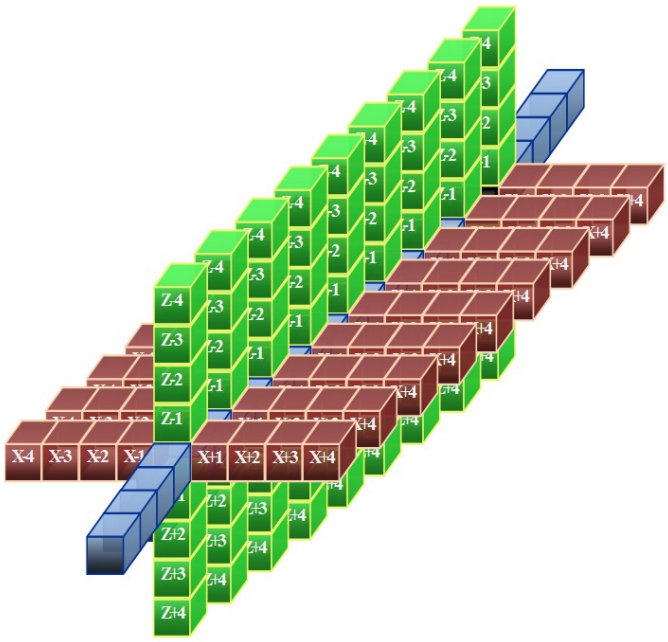

Figure 6: 3D Stencil Vector Access

used, the supported memory access patterns for multiple vector instructions are discussed in this section. The EMVS uses GNU gcc 4.2.0 compiler. We implement C macros which facilitate the programming of general access patterns and loop generation by a set of function calls, combined with an API. The data transfer and multi VP cores processing information are included in the EMVS header file and implements function calls (e.g. STRIDE(), INDEXED(), etc.) that require primary knowledge of the local memory and the data set. The programmer has to define the code using EMVS function calls. The function calls are used to transfer the whole data set between the *Main Memory* and the *Specialized Vector Memory*. EMVS manages complex data transfers in parallel with vector processing.

3.2.1. Task Scheduling

Like generic multi-core system the EMVS *task*s are programmed at program-time. Unlike other systems, the EMVS *Scheduler* performs the task scheduling in hardware. The EMVS *Programming Model* takes the parallel program, converts the program into multiple tasks and places the information of the task in *Scheduler* buffer. At run-time, the EMVS executes the multiple tasks on multiple VP cores independently by using the EMVS *Scheduler*. The *Scheduler* buffer

perform pipelining and divides the elements of work, requested by the vector instruction into smaller groups that are mapped onto the VL lanes of a single or multiple VP cores. The *Scheduler* generates a hazard if two or more of parallel vector instructions conflict and do not execute in consecutive clock cycles. The *Scheduler* checks the hazard for the VP cores and generates flags and stalls if required. The parallel task execution process is performed in three stages where VL operand pairs are read from the register file and sent to the functional units in the VL lanes.

3.2.2. Data Transferring

Figures 3 (a), (b) and (c) show vector loops (with MVL of 64) for a scalar processor architecture, conventional vector architecture and the EMVS, including the EMVS memory transfer instructions respectively. The `VLD.S` instruction transfers data with the specified stride from the *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 and efficient prefetching support. Delay and power increase for complex non-stride accesses and crossbars. The `EMVS_VLD` instruction uses a single or multiple descriptors to transfer data from the *Main Memory* to a *Specialized Vector Memory*. EMVS rearranges and manages accessed data in the *Buffer Memory* and transfers it to vector registers. In Figure 4, EMVS prefetches vectors longer than MVL in the *Specialized Vector Memory*. After completing the first transfer of MVL, the EMVS sends a signal to the vector processor that acknowledges that the register elements are available for processing. In this way, EMVS pipelines the data transfers and parallelizes computation, address management, and data transfers.

The EMVS memory manager efficiently transfers data with long strides, longer than MVL size and feeds it to a VP core. 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 5. By combining these descriptors, the EMVS exchanges 3D data between the *Main Memory* and the *Specialized Vector 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 the direction of row, column and plane respectively. The x, y and z vectors have a 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. $row\_size \times column\_size$. For multiple or complex vector

accesses (shown in Figure 6), EMVS prefetches data using vector access function calls (e.g. INDEXED(), etc.), arranges them according to the predefined patterns and buffers them in the *Specialized Vector Memory*.

4. Experimental Framework

In this section, we describe the FPGA based Embedded Multi Vector-core Processor, GPU and Heterogeneous Multi-core Systems. The section is further divided into five subsections: the *Odroid XU3 System*, the *Parallela System*, the *Jetson TK1 System*, the *FPGA based EMVS* and the *Test Applications*.

4.1. Odroid XU3 System

The *Odroid XU3 System* (shown in Figure 7) uses embedded heterogeneous multi-core architecture. The architecture has Octa-Core processing system which gives high performance with less energy consumption The processing system consists of a high-performance a Cortext-A15 quad-core processor, a low-power Cortex-A7 quad-core processor and an ARM Mali-T628 GPU processor with six cores. The system uses 2GB DRAM LPDDR3 as main memory. The Odroid System contains four real-time current sensors for measuring dynamic power. The ARM A7 and A15 processors operate the same ISA with different performance features. Ubuntu 14.04 Operating System and C and C++ programming languages are used to program the applications kernels by using g++ compiler version 4.9.2, with the support of OpenCL v1.1, OpenMP 4.0 and Pthreads. The board also includes 30 general purpose input outputs (GPIO) which include parallel I/O, SPI, I2C, ADC, UART and GPIO IRQ are used to interface the real-time embedded applications.

4.2. Parallella System

An open source computer platform called Parallella is also used (shown in Figure 8). The *Parallella System* is a heterogeneous multi-core, high performance computing system. The heterogeneous architecture of *Parallella System* uses a Zynq System-On-Chip (SoC) and Adapteva accelerators. The Zynq SoC uses dual-core ARM-A9 processors and Artix-7 FPGA having 85K logic cells. The Adaptevas Epiphany uses16-multi-accelerators. Each accelerator uses a 32-bit dual-issue superscalar RISC architecture working at 1 GHz of the clock frequency. The accelerator core can deliver maximum performance of 32 GFLOPS. The cores are connected by using a 2D-mesh Network on a Chip (NoC) bus system. The accelerator cores support floating-point RISC ISA and have a shared global address

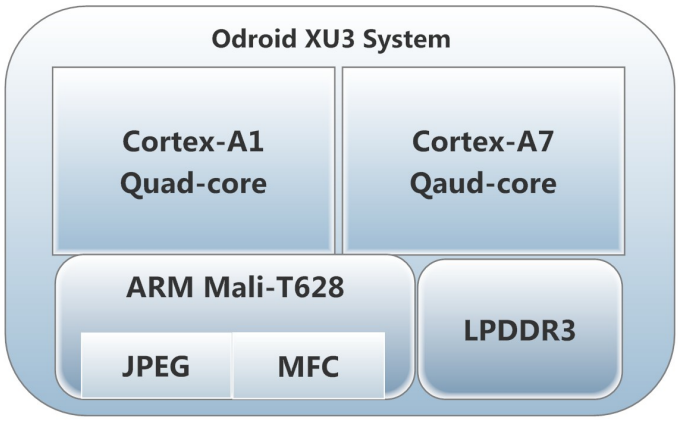

Figure 7: Heterogeneous Multi-core: Odroid XU3 System

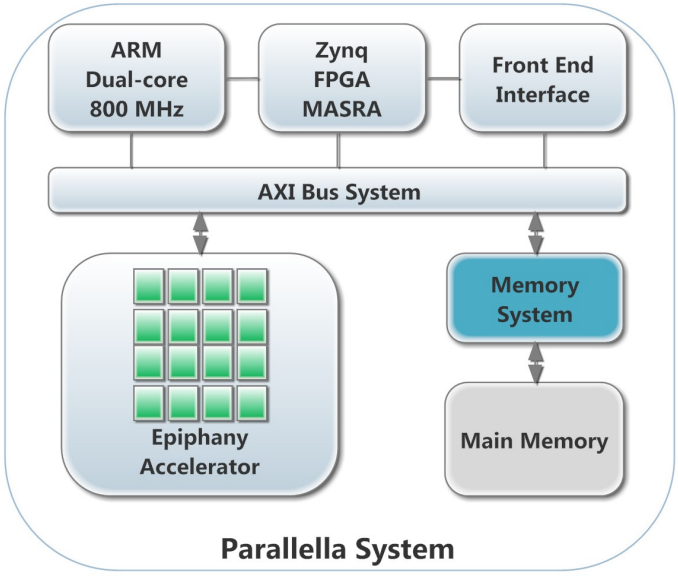

Figure 8: Multi-core: Parallella System

space. To program the applications OpenMP C infrastructure is used. The *Parallella System* OpenMP is a source to source compiler and provides runtime system flexibility. The compiler takes an application written in C annotated with OpenMP pragmas.

4.3. Jetson TK1 System

The *Jetson TK1 System* is a GPU based embedded supercomputer board shown in Figure 9. The Jetson TK1 is integrated by a Tegra K1 processing system, which gives high performance computing with low power for embedded systems applications.

The *Jetson TK1 System* is a CUDA-capable GPU based mobile processor using a GK20A Kepler GPU architecture with 192 single precision CUDA cores and a quad-core ARM Cortex-A15 processor. The *Jetson TK1 System* uses dual Image Signal Processor (ISP) Core that delivers 1.2 GigaPixels per second of raw processing power supporting camera sensors up to 100 Megapixels. The *Jetson TK1 System* has 2GB of main memory. Ubuntu 14.04 distribution is used to operate the *Jetson TK1 System*. The applications are programmed C++ programming language and accULL compiler. The accULL is a free compiler that works on ARM processors. The accULL distributes the compute intensive part of the program on GPU cores. It translates the compute-intensive part into CUDA code using directives similar to OpenACC.

4.4. FPGA based EMVS

The *FPGA based EMVS* architecture is shown in Figure 10. The *FPGA based EMVS* design uses multiple VP cores each core has parameterizable design enabling a large design space of possible VP configurations. These parameters can modify the EMVS processing system architecture, instruction set architecture, and memory system. In the current FPGA evaluation on Altera DE4 board, four VP cores are integrated into the *Multi-Vector System*. Two VP cores use 32 lanes, and

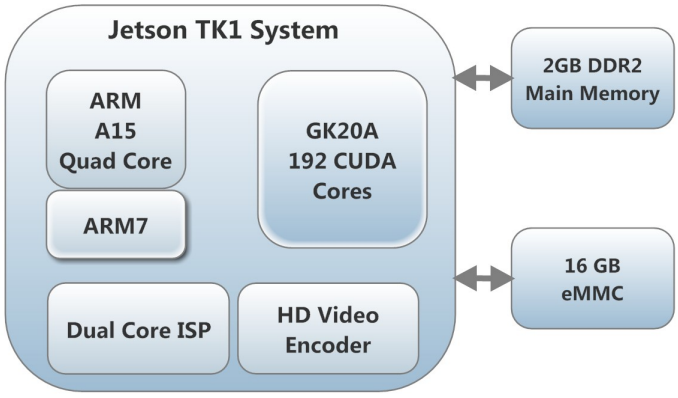

Figure 9: GPU based: Jetson TK1 System

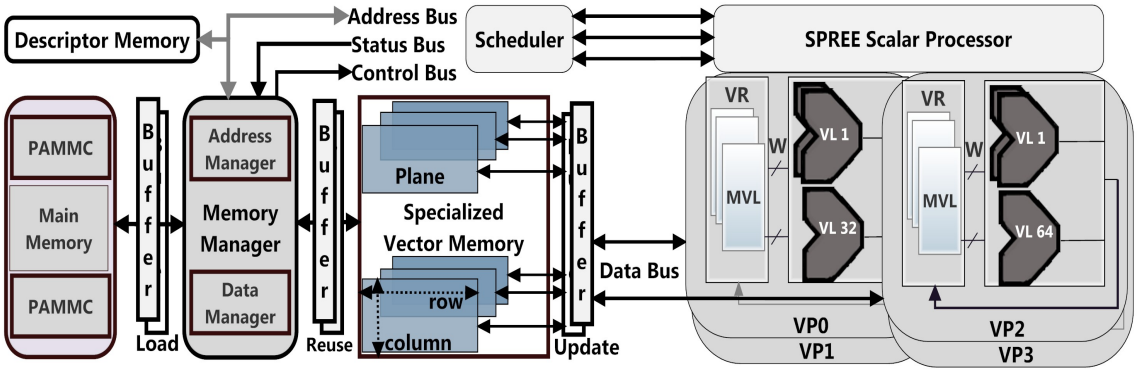

Figure 10: Altera Stratix-IV FPGA based EMVS Architecture

other two are using 64 lanes. Each VP core uses 8kB of SVM as local memory. 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.

An SPREE [16] scalar processor is used to execute the scalar operations of *Multi-Vector System*. 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 except for control instructions.

The *FPGA based EMVS* compiler performs the task partitioning, assign inter VP core communication and maps the tasks to the *Scheduler*. The priorities of each vector processor core task can statically declare. From the main program, the application is spawned as multiple parallel threads. At run-time, the EMVS scheduler handles multiple VP cores processing tasks and manages on-chip data and off-chip data movement using the *Buffer Memory* and the *Descriptor Memory*.

4.5. Applications

Table 1 shows the problem applications that are processed on the systems (discussed above) along with their required number of processing. The applications table incorporates a broad range of processing and data transfer requirement. These requirements are used to measure the behavior and performance of data processing, data management and data transfer of the systems in a variety of situations. Column *Vector Operation* presents maximum number vector operation used by an application. Each application uses a multiple processor cores.

5. Results and Discussion

In this section, the system performance, dynamic power and energy of the *Odroid XU3 System*, the *Parallela System*, the *Jetson TK1 System* and the *FPGA based EMVS* is measured by executing *Test Applications* (shown in Table 1) at 200 MHz of clock rate. The section is further divided into the *Application Performance Comparison*, the *System Performance Comparison* and the *Dynamic Power and Energy Comparison*

5.1. Application Performance Comparison

For performance comparisons, we execute the applications of Table 1 with 256 MB of dataset on embedded systems (discussed in Section 4. Figure 11 shows the

Table 1: Brief description of application kernels

Application	Description	Vector Operation
FIR	Calculates the weighted sum of the past and current inputs	32
1D Filter	Low pass 1D Filter	32
Tri-Diogonal	Determines the best local arrangements among nucleotide or protein patterns	32
Mat_Mul	Performs Matrix Multiplication X=Y $\times$ Z	64
RGB2Gray	Converts 24bit RGB to 8bit Gray scale image	32
RGB2CMYK	Converts RGB image data into CMYK format	32
Gaussian	Implements discrete convolution filter to estimated the 2nd order derivatives	64
Image Blend	Blend two images and generate one image file	32
K-Mean	A method of vector , quantization perform cluster analysis in data mining.	32
3D-Stencil	Averages the nearest neighbor points in Three Dimensions	96

applications processing time of the *Odroid XU3 System*, the *Parallela System*, the *Jetson TK1 System* and the *FPGA based EMVS*. The X-axis represents application kernels. The Y-axis shows the application execution time (seconds) in logarithmic scale (less is better). Each bar represents the application data transfer time and processing time for the embedded systems.

By using the FPGA based EMVS, the results show that the FIR kernel achieves 12.69x 9.94x and 5.24x of the speedup over the *Odroid XU3 System*, the *Parallela System* and the *Jetson TK1 System* respectively. The 1D Filter application achieves 14.05x, 6.5x and 3.19x of speedups. The FIR and 1D Filter application kernel data processing have no data dependencies, therefore EMVS uses multiple descriptors to access data and feed them multi-vector processor for the processing. The Tri-diagonal kernel processes the matrix with sparse data placed in diagonal format. The application has a diagonal data transfers with less data locality, therefore attains 6.64x, 4.84x 2.69x of speedup. The Mat_Mul kernel accesses row and column vectors. EMVS uses two descriptors to locate the two vectors. The row vector access pattern has unit stride whereas the column vector has a stride equal to the size of a row. The EMVS gets 7.46x 4.63x 3.14x of speedups. RGB2CMYK and RGB2Gray have a 1D block of data access with maximum data locality and achieves 14.63x, 6.51x, 3.89x and 19.28x, 6.37x, 4.46x of speedups respectively. The Gaussian and Image Blend applications take the 2D block of data perform 2D

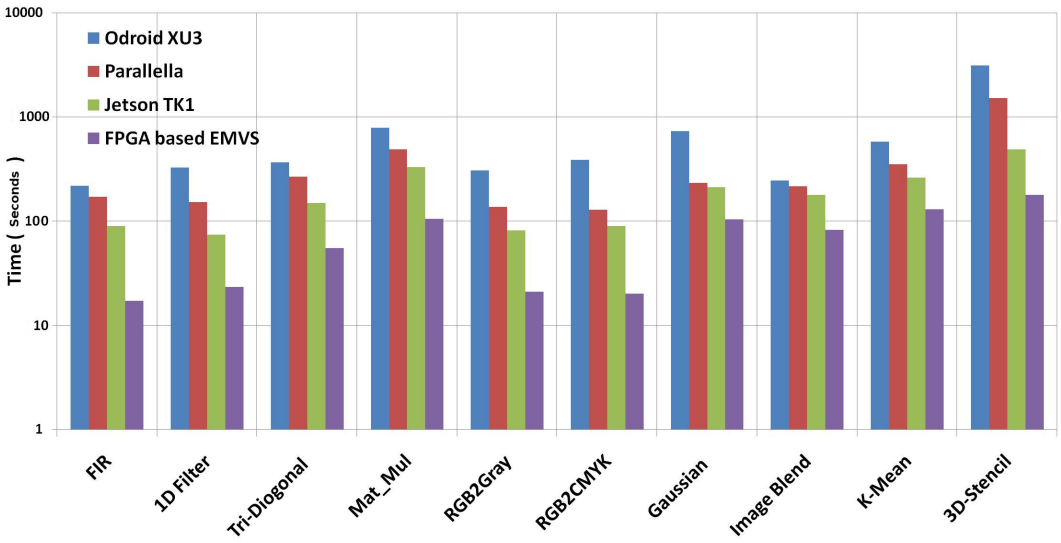

Figure 11: Single Application Execution Time on Different Vector Systems

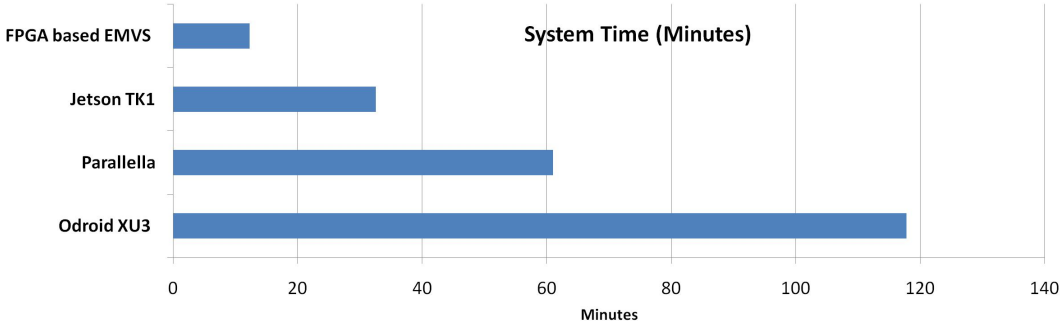

Figure 12: Embedded Systems Time in Minutes while Processing the Applications Concurrently

processing. The applications achieve 7.08x, 5.40x, 2.28x and 8.40x, 4.90x, 2.94x of speedups respectively. The K-Mean kernel has random load/store and 1D stridden data transfers with no data locality. The EMVS achieves 4.48x, 2.71x, and 1.46x of speedups. The reason for fewer speedups is, the application has random memory accesses for the processing and has multiple branches, the EMVS uses the scalar processor to perform the processing. The 3D-Stencil uses three-dimensional data structure. The EMVS *Specialized Vector Memory* efficiently handle 3D and feed it to multiple VPs. The EMVS system achieves 17.53x, 8.54x and 4.12x of speedups. The vectorized applications of the *Odroid XU3 System*, the *Parallela System* and the *Jetson TK1 System* always use the maximum processing cores and the memory system accesses data with unit stride using load/store and DMA operations. The memory management of the system uses a multi-banking methodology which requires larger crossbar to routes the data to multiple processing cores. The EMVS manages memory access patterns of multiple vector processors in hardware thus improves the system performance by prefetching these complex access patterns in parallel with computation and by transferring them to vector processors without using a complex crossbar network. The EMVS *Specialized Vector Memory* unit holds complex vector data structures and efficiently accesses, reuses, aligns and feeds data to multiple vector processors. EMVS supports multiple data buses that increase the local memory bandwidth and reduce on-chip bus switching.

5.2. System Performance Comparison

The system performance is measured by executing application kernels simultaneously, on the *Odroid XU3 System*, the *Parallela System*, the *Jetson TK1 System* and the *FPGA based EMVS*. Figure 12 shows the systems total time in minutes

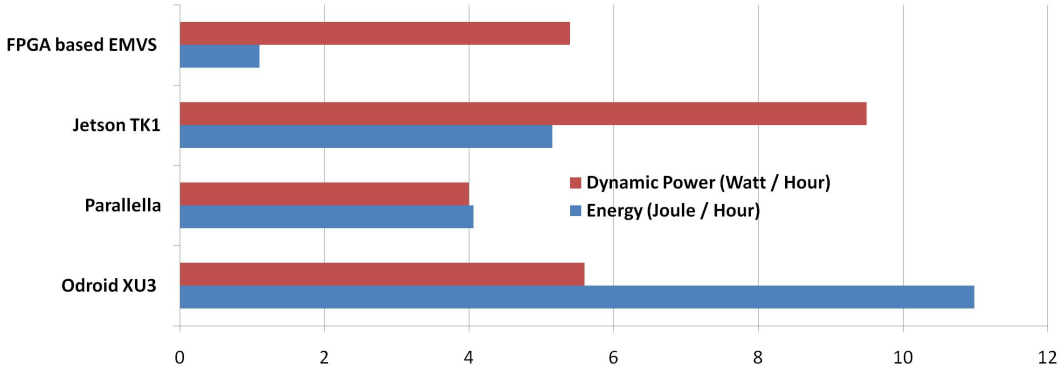

Figure 13: Dynamic Power and Energy of Embedded System Architectures

while processing the application kernels concurrently. Each bar includes the applications parallel processing time on multiple processors, data transfer and management time and application tasks scheduling time. While processing the application on the EMVS, the results show that the EMVS achieves 10.22x, 5.67x and 3.01x of performance improvement against the *Odroid XU3 System*, the *Parallela System* and the *Jetson TK1 System* respectively. The EMVS manages memory access patterns in hardware which improve the system performance by prefetching these complex access patterns in parallel with the vector processing. The EMVS *Scheduler* applies run-time resource-aware scheduling policies which efficiently executes the application tasks on multiple processing cores.

5.3. Dynamic Power and Energy Consumption

The dynamic power and energy of the *Odroid XU3 System*, the *Parallela System*, the *Jetson TK1 System* and the *FPGA based EMVS* is measured (shown in Figure 13), while executing the *Test Applications* 1. To measure dynamic power the DE4 board gives a resistor to sense current/voltage and 8-channel differential 24-bit analog to digital converters. While executing the test application, the results show that the *Odroid XU3 System*, the *Parallela System*, the *Jetson TK1 System* and the *FPGA based EMVS* draw 5.6, 4.1, 9.5 and 5.4 watts of dynamic power respectively. While comparing the dynamic power results show that EMVS consumes 1.04 and 1.76 times less power than the *Odroid XU3 System* and the *Jetson TK1 System* respectively and draws 1.35 times more power than the *Parallela System*. Whereas when we compared the energy it is absorbed that the EMVS consumes 10.60, 4.20 and 5.30 times less energy than the *Odroid XU3 System*, the *Parallela System* and the *Jetson TK1 System*.

6. Related Work

Severance et al [17] presents the VectorBlox MXP Matrix Processor, an FPGA-based soft processor capable of highly parallel execution. The MXP processor system utilizes C languages to program and executes data- parallel software algorithms in hardware. The MXP processor architecture uses custom vector instructions and expandable DMA filters. The processor is integrated into existing Altera and Xilinx development flows which simplifies the development process. Severance et al. [18] also developed a pipelining data path technique for the Streaming Vector Processor (SVP). The SVP handles the pipeline data with high-bandwidth and manages the outputs in an on-chip memory. The SVP processor uses C language to programme the vector applications. Codreanu et al. [19] proposed an adapted interleaved multi-threading technique that improves the performance and reduces the vector processor energy consumption. The technique improves the scalar processor pipeline performance and increases vector resources utilization.

Yu et al. [20] suggest a uni-core vector processor architecture called VIPERS. The VIPERS consists of a RISC scalar core, that performs the memory management, an address generation logic, and a memory crossbar that control the data transfer movements and a vector core for processing application. Chou et al. [21] provide a vector processor architecture having a scratchpad memory for local data management called VEGAS. Severance et al. [22] proposed an updated version of VEGAS called VENICE. The VENICE uses a scratchpad for software-based data management and a DMA to access data from the main memory. VENICE has reservations about the rearrangement of irregular/complex data with scatter/gather support. Yiannacouras et al. propose the VESPA [23] processor that utilizes a configurable cache and provides prefetching support in hardware for a fixed number of cache lines that increase the memory system performance. The VESPA system integrates wide processor buses to meet the system cache line sizes. VIPERS and VEGAS integrates a scalar Nios core that moves the data between the main memory and the local memory. To align and rearrange the local data, the systems use a crossbar network.

Robert et al [24] developed a single-board computer (SBC) at Los Alamos National Laboratory for the aeronautical applications. The SBC is intended to adhere the data and command handling demands for space missions by applying true space-grade radiation hardness and fault tolerance. The design uses a space-quality dual-core application specific integrated circuit (ASIC) processor, an FPGA, memories, and front-end analog and digital interfaces to meet the command- and data-handling requirements of medium-sized missions. The design

consumes 5 W and measuring less than 7 inches x 6 inches, the design supports 9-gigabit/s class bidirectional SerDes links, 6 SpaceWire ports, redundant MIL-STD-1553B ports, 32 Mbytes of EDAC protected SRAM, 2 GBytes of nonvolatile memory, and supports 200 MFLOPS operation.

Inoue [25] proposed a multi-vector processor system that largely aims at efficient satellite image processing. The multi-vector processor system has up to 64 processor units, a loop network, and an image memory. The processor units can execute flexible vector processing with a unique vector access arrangement. The loop network produces high-speed and contention-free data transfer among the processor systems. Ronny [26] proposed the vector-thread (VT) architecture, which provides high performance with little power and small area. The VT structure unites the vector and multithreaded compute models. It uses a control processor that fetches commands and broadcast instruction to all vector processor. The vector and threaded control mechanisms provide a VT architecture to flexibly and compactly encode application parallelism and locality, and a VT machine utilizes to gain performance and effectiveness. Ronny [27] also proposed a vector-thread architectures as a performance-efficient solution for all-purpose computing. The VT structural model joins the vector and multithreaded compute designs. VT gives the programmer with a command processor and a vector of virtual processors. Mike [28] described the SunOS multi-threading architecture for multiple threads. The SunOS uses the lightweight threads to increase the UNIX Application Programming Interface for a multi-threaded environment. The design permits the programmer to check the level of physical concurrency the application needs or permits the threads package to determine automatically this. The SunOS architecture applies a regular synchronization mechanism between the threads to handle local and global processes. The programmer can manage the mapping of threads onto LWPs to obtain special improvement.

Tassadaq et al. [29] [10] discussed a uni-vector accelerator based memory system and its implementation on an Altera FPGA to establish a fast communication with the host. The design supports application-specific accelerators and scalar soft core processor and integrates a pattern based Memory Controller for Vector Processor. The Memory Controller for Vector Processor uses *Buffer Memory* and a *Data Manager* are integrated that reduce the power dissipation and efficiently access, reuse, align and feed data to the vector processor without a complex crossbar network.

7. Conclusion

In this work, we proposed and designed an Embedded Multi Vector-core System (EMVS) architecture, that manages and schedules multiple vector processor and data transfers. The EMVS consists of parameterizable vector processors (VP), vector lanes (VL), specialized vector memory, on-chip bus network and Pattern Aware Main Memory Controller. The system uses C macros that applies loop unrolling and strip-mining and access complex vector data for multiple vector processors. The EMVS memory manager and scheduler handles multi vector data transfer and scheduling instruction in hardware. The performance of EMVS is compared with different heterogeneous multi-core system architectures. The benchmarking results show that EMVS achieves between 2.7x to 9.6x of speedup against the GPU and Heterogeneous Multi-core Systems.

References

[1] Earl Swartzlander. *Application Specific Processors*, volume 380. Springer Science & Business Media, 2012.

[2] Kevin W Rudd. *Vliw processors: efficiently exploiting instruction level parallelism*. PhD thesis, Citeseer, 1999.

[3] Rem Gensh, Ali Aalsaud, Ashur Rafiev, Fei Xia, Alexei Iliasov, Alexander Romanovsky, and Alex Yakovlev. Experiments with odroid-xu3 board. *Newcastle University, Computing Science, Claremont Tower, Claremont Road, Newcastle England*, 2015.

[4] Spiros N Agathos, Alexandros Papadogiannakis, and Vassilios V Dimakopoulos. Targeting the parallella. In *Euro-Par 2015: Parallel Processing*, pages 662–674. Springer, 2015.

[5] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. The stratix 10 highly pipelined fpga architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 159–168. ACM, 2016.

[6] 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.

[7] Xilinx Virtex-7. Leading fpga system performance and capacity, 2012.

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

[9] Tassadaq Hussain et al. Reconfigurable Memory Controller with Programmable Pattern Support. *HiPEAC WRC*, Jan, 2011.

[10] Tassadaq. Hussain et al. PPMC: A Programmable Pattern based Memory Controller. In *ARC 2012*.

[11] T. Hussain, O. Palomar, A. Cristal, O. Unsal, E. Ayguady and M. Valero. Advanced Pattern based Memory Controller for FPGA based Applications. In *International Conference on High Performance Computing & Simulation*, page 8. ACM, IEEE, 2014.

[12] T. Hussain, O. Palomar, A. Cristal, O. Unsal, E. Ayguady and M. Valero. Advance Multi-core Memory Controller. In *International Conference on Field-Programmable Technology FPT2014*. IEEE, 2014.

[13] T. Hussain, O. Palomar, A. Cristal, O. Unsal, E. Ayguady and M. Valero. MAPC: Memory Access Pattern based Controller. In *24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014.

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

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

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

[17] Aaron Severance and Guy GF Lemieux. Embedded supercomputing in fpgas with the vectorblox mxp matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[18] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 117–126. ACM, 2014.

[19] Valeriu Codreanu, Lucian Petrică, and Radu Hobincu. Increasing vector processor pipeline efficiency with a thread-interleaved controller. In *System Theory, Control, and Computing (ICSTCC), 2011 15th International Conference on*, pages 1–4. IEEE, 2011.

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

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

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

[23] Yiannacouras, and others. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *CASES 2008, Proceedings of international conference*.

[24] Robert Merl and Paul Graham. A low-cost, radiation-hardened single-board computer for command and data handling. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2016.

[25] Atsushi Inoue and Akira Maeda. The architecture of a multi-vector processor system, vpp. *Parallel Computing*, 8, 1988.

[26] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. The vector-thread architecture. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. IEEE, 2004.

[27] Ronny Krashinsky. *Vector-thread architecture and implementation*. PhD thesis, Massachusetts Institute of Technology, 2007.

[28] Mike L Powell, Steve R Kleiman, Steve Barton, D Shan, Dan Stein, and Mary Weeks. Sunos multi-thread architecture. In *The SPARC Technical Papers*. Springer, 1991.

[29] T. Hussain, O. Palomar, A. Cristal, O. Unsal, E. Ayguady and M. Valero. Memory Controller for Vector Processor. In *The 25th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE ASAP 2014 Conference, 2014.