AMC: Advanced Multi-accelerator Controller

[1,2],Tassadaq Hussain, [3],Amna Haider, [3]Shakaib A. Gursal, and [1]Eduard Ayguadé

[1] *Barcelona Supercomputing Center* , [2]*BSC-Microsoft Research Centre* ,
[3]*Unal Center of Engineering Research & Development*
Barcelona, Spain
{*tassadaq.hussain, eduard.ayguade*} *@bsc.es*
{*first name*}*@ucerd.com*

Abstract

The rapid advancement, use of diverse architectural features and introduction of High Level Synthesis (*HLS*) tools in FPGA technology have enhanced the capacity of data-level parallelism on a chip. A generic FPGA based *HLS* multi-accelerator system requires a microprocessor (master core) that manages memory and schedules accelerators. In a real environment, such *HLS* multi-accelerator systems do not give a perfect performance due to memory bandwidth issues. Thus, a system demands a memory manager and a scheduler that improves performance by managing and scheduling the multi-accelerator's memory access patterns efficiently. In this article, we propose the integration of an intelligent memory system and efficient scheduler in the HLS-based multi-accelerator environment called Advanced Multi-accelerator Controller (AMC). The AMC system is evaluated with memory intensive accelerators, High Performance Computing (HPC) applications and implemented and tested on a Xilinx Virtex-5 ML505 evaluation FPGA board. The performance of the system is compared against the microprocessor-based systems that have been integrated with the operating system. Results show that the AMC based HLS multi-accelerator system achieves 10.4x and 7x of speedup compared to the MicroBlaze and Intel Core based HLS multi-accelerator systems.

1. Introduction

In the last few years the density of FPGAs [1, 2] and performance per watt [3] have improved, which allows the High Performance Computing (HPC) industry to increase and provide more functionalities on a single chip. 3D ICs [4] open

another dimension in the HPC industry that emulate three-dimensional stacked chips by rapidly reconfiguring their two-dimensional fabrics in a third spatial dimension of time/space. Such devices have the power to reconfigure their fabric up-to 1.6 billion times per second [4]. The Stacked Silicon Interconnect (SSI) [5] technology provides another dimension to high-density FPGAs that satisfies the needs of on-chip resources for HPC system. SSI combines two or more FPGAs for larger and complex systems. As the designs grow larger and complex, the chances of error and complexity increase, thus demands an abstract level design methodology.

Time is indeed changing the way HPC systems are designed. Now the HPC industry wants to execute multi-applications for optimal performance. This demands a design environment which has dense and flexible hardware consumes less power and has abstract-level programming tools. Reducing design time and the size of the team are crucial. Numerous architectures (RISC, CISC), design methodologies (SoC, MPSoc), and programming tools (OpenMP, MPI) are available in the HPC domain. These systems do not give a perfect performance due to the processor-memory speed gap [6, 7] given by the formula

$AT = P_{rob} * On\,Chip\,Memory + (1 - P_{rob}) * [On/Off_{chipbus} + DRAM]_t.$

The Access Time *(AT)* depends upon the probability (P_{rob}) to access data from different units such as $On\ Chip\ Memory$ (On-Chip data arrangement and probability of reuse), $on/off_{chipbus}$ (flow control, arbitration, translation, interconnection) and $DRAM_t$ (bank, column and row selection).

Over the past few years, HLS tools have been strengthened and become truly production-worthy [8] by supporting fundamental technologies such as pipelining and FSM. Initially confined to data path designs, HLS tools are now commenced to address complete systems, including control logic, complex on-chip and off-chip interconnections. HLS [9] tools provide design modeling, synthesis, and validation at a higher level of abstraction. This makes computation cheaper, smaller, and less power hungry, thus enabling more sophisticated system design for complex HPC applications. HLS-based multi-accelerator systems always require a master processor core which perform memory management, data transfer and scheduling of the multi-accelerator. The master core adds overhead of address/data management, bus request/grant and external memory access time.

To improve HLS-based multi-accelerator system performance an intelligent controller is needed that has efficient on-chip *Specialized Memory* and data management, an intelligent front-end/back-end scheduler and a fast I/O link and supports programming model that manages memory accesses in software so that hardware can best utilize them. We implement such a controller in hardware called

Advanced Multi-accelerator Controller (AMC), some salient features of the proposed AMC architecture are given below:

- The AMC based system can operate as a stand-alone system, without support of the master cores and operating system (OS).
- The AMC provides specialized on-chip (1D/2D/3D) memory that improves system performance by prefetching complete data sets.
- Complex access patterns are described using single or multiple descriptors, thus reduces the on-chip communication time.
- It handles multiple HLS hardware accelerator IPs using an event-driven handshaking methodology; this decreases the time-to-market and complexity of hardware.
- The AMC schedules multi-accelerator requests while taking into account both the processing and memory requirements defined at program-time. At run-time, the AMC back-end scheduler reorders accelerator's memory requests considering SDRAM open banks. This removes the overhead from opening/closing rows/banks and idle cycles on the data bus.
- Standard C/C++ language calls are supported to program multi-accelerator tasks in software.

Section 2 discusses the HLS-based system. Section 3 describes the AMC system and Section 4 explains how it is used. Section 5 introduces the experimental framework and Section 6 presents the results. Section 7 discusses the related work and finally Section 8 provides the conclusions.

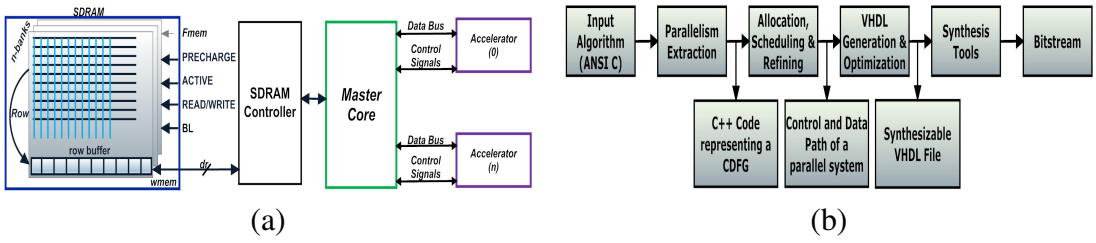


Figure 1: (a) Generic High Level Synthesis Multi-Accelerator System (b) High Level Synthesis Tool Design Flow

2. High Level Synthesis Multi-accelerator System

In this section, we discuss High-Level Synthesis (HLS) system design tools and working operation. The HLS tool takes algorithm level behavioral specification of a system and generates a register-transfer level structure that implements that behaviour. The HLS-based multi-accelerator system is shown in Figure 1 (a). The HLS system is further categorized into three sections the Master Core, the HLS tools and the SDRAM controller.

2.1. Master Core

A Master Core (CPU) is capable of initiating data transfer on the bus for multi-accelerator (slave). The core is responsible for memory management and scheduling of HLS multi-accelerator system. It decides when an accelerator start execution and provides a I/O link. All accelerators can communicate with each other via Master core. The HLS system does not get optimum performance if accelerators are not tightly scheduled.

2.2. High Level Synthesis Tool

In this section, we explain existing HLS tool and their basic operation. Translation of HPC algorithms into hardware is difficult and time-consuming. Large application result in big systems with higher chances of failure and complexity in the scheduling and reuse of resources. To control the size and complexity of such applications, system designers are required to perform design modeling, synthesis, and validation at a higher level of abstraction. Ylichron (now PLDA Italia) provides a source-to-source C to VHDL compiler toolchain targeting system architects called HCE (Hardware Compiling Environment) [10]. ROCCC 2.0 [11] is a free and open source tool that focuses on FPGA-based code acceleration from a subset of the C language. ROCCC tries to utilize parallelism within the constraints of the target device, optimize clock cycle time by pipelining, and minimizes area.

A generic HLS toolchain takes ANSI-C language as input, which represents the hardware architecture with some restrictions and extensions. The design flow consists of a series of steps (see Figure 1 (b)), with each step transforming the abstraction level of the algorithm into a lower level description. The HLS tool first extracts the Control and Data Flow Graph (CDFG) from the application to be synthesized. The CDFGs describe the computational nodes and edges between the nodes. Tool provides different methods to explore the CDFG of the input algorithm and generates the data-path structure. The generated data-path structure contains the user-defined number of computing resources for each computing

node type and the number of storage resources (registers). The *Allocation and Scheduling* step maps the CDFG algorithm onto the computing data-path and produces a Finite State Control Machine. The *Refine* step uses the appropriate *Board Support Package* and performs synthesis for the *communication network*. Finally, the VHDL-RTL generation step produces the VHDL files to be supplied to the proprietary synthesis tools for the targeted FPGA.

2.3. SDRAM Controller

In this section, we describe basic parameters and working operation of Synchronous Dynamic Random Access Memory (SDRAM). SDRAM is dynamic random access memory (DRAM) that is synchronized with the processor system bus (shown in Figure 1 (a)). The parameters *nbanks, wmem, fmem, dr*, and *BL* are used to describe the SDRAM memory architecture. Where `nbanks` stands for the number of banks, `wmem` is the width of the data bus in bytes, `fmem` represents the clock frequency of memory in MHz, `dr` defines the number of data words that can be transferred during a clock cycle, and `BL` is the word length of programmed burst. To control the SDRAM system, specific commands are sent to the memory port according to the protocol. The SDRAM communications protocol includes six command signals which are activate (ACT), read (RD), write (WR), precharge (PRE), refresh (REF), and no-operation (NOP). The activate command is supplied with a row and a bank as an argument, informs the selected bank to copy the requested row to its row buffer. Once the requested row is opened, column can be accessed by read and write bursts having burst length (`BL`) of 4 or 8 words. The read and write commands include separate bank, row, and column address lines. The precharge command corresponds the activate command to place the contents of the row buffer back to its location in the memory. An auto-precharge flag with a read and write command is inserted to automatically precharge at early possible moment after the data transfer. This gives the benefits to an upcoming scheduling row to be opened as quickly as possible without causing contention on shared system bus. To prevent the data loss by capacitor leakage, a refresh command must constantly be issued. For entire memory array, multiple refresh commands are required.

3. Advanced Multi-Accelerator Controller

In this section, we describe the Advanced Multi-Accelerator Controller (AMC) system (shown in Figure 2) for HLS tool. The architecture is based on five units:

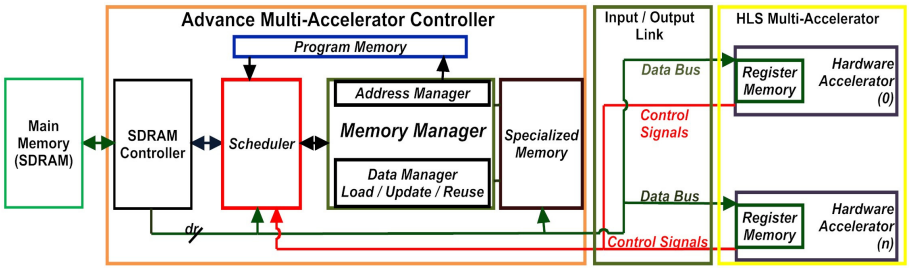


Figure 2: Architecture of Advanced Multi-Accelerator Controller Based System

the *Input/Output link*, the *Memory Architecture*, the *Memory Manager*, the *Scheduler* and the *SDRAM Controller*.

3.1. Input/Output Link

The *Input/Output (I/O) link* provides an interface between AMC and HLS multi-accelerator unit. The *I/O link* as shown in Figure 2 is used to read/write high-speed data to/from register memory of HLS multi-accelerator. Transfer of data is accomplished according to the system clock. The AMC *Input/Output (I/O) link* has a peak bandwidth of 1.6 GB/s as it operates at maximum 200 MHz clock having data bus width of 64-bit. A *state controller* (Figure 9 (b)) is integrated with each hardware accelerator to manage multiple buffers. The state controller takes accelerator data *requests* and manages multiple buffers using *request* and *grant* signals.

3.2. Memory Architecture

A reliable and efficient memory architecture is required for any application being programmed for HLS multi-accelerator system. The AMC memory architecture is divided into two sections the *Descriptor Memory* and the *Data Memory*.

3.2.1. Descriptor Memory

The AMC descriptor memory holds the information of multi-accelerator memory access patterns and scheduling methodology. The descriptor memory is fixed length data word with a number of attribute fields that describe the access pattern. A single descriptor describes the access pattern of a strided stream, and multiple descriptors are used to define complex memory patterns. The least set of parameters for the memory descriptor block, includes command, source address, destination address, stride, stream and priority. Command specifies whether to

```
receive (/*Local Address*/0x00000000, /*External Address*/0x00000100,
/*Priority */0x00, /*Stream*/0x08, /*Stride */0x00);
```

Figure 3: AMC Function Call: Single Descriptor Program

read/write a single element or a stream of data. The address parameters specify the starting addresses of the source and destination locations. Stride indicates the distance between two consecutive physical memory addresses of a stream. The priority describes the selection and execution criteria of an accelerator. The program structure of AMC is stack based where descriptor holds information of complete memory access pattern. The machine code of AMC consists of just an opcode that activate/deactivate accelerator kernel. Memory management and optimization for repeated memory access is done at compile time and are executed in hardware at runtime. The proposed AMC system provides C and C++ language support that initializes descriptor blocks. The program example of a single memory access is presented in Figure 3. The *local address* and *external address* parameters hold the start address of specialized (*buffer*) memory and main memory (*SDRAM*) data set respectively. The *Priority* defines the order in which memory access is entitled to be process. The parameters *Stream* and *Stride* define the type of memory access. The value `0x08` and `0x00` of *Stream* and *Stride* parameters respectively initializes AMC to access a stream of 8-words with a unit stride. At run-time, the AMC accesses 32-bit word from the `0x00000100` address location of main memory and write it to the `0x00000000` address location of *Specialized Memory*. The following memory access till the end of a stream, the AMC generates main memory address by taking the address of main memory from the previous access and adds *stride* (`0x08`) value. The *Specialized Memory* address is sequential (contiguous) and requires unit increment.

3.2.2. Data Memory

To achieve high performance from any computing unit, analysis of the data transfer through the algorithm and placement on underlying memory architecture is required. The AMC underlying data memory structure is shown in Figure 5, The data memory of AMC system is further subdivided into three sections which are: the *Register Memory*, the *Specialized Memory* and the *Main Memory*.

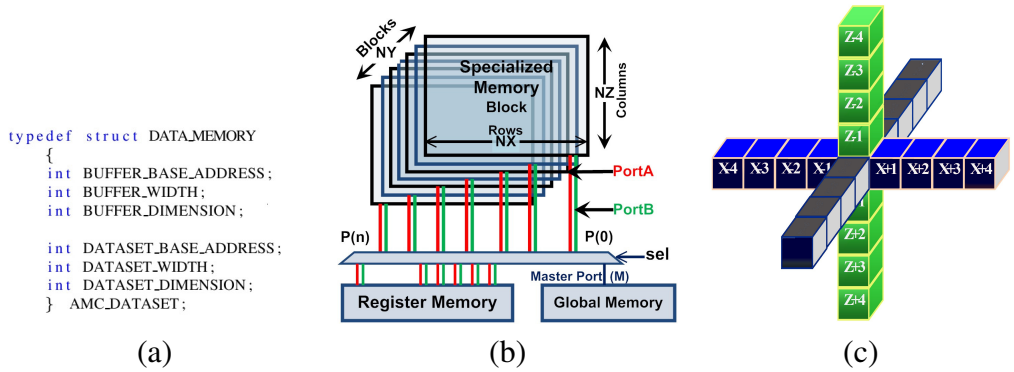


Figure 4: AMC: (a) Dataset/Program Code (b) Three Dimension Specialized Memory Architecture (c) (3 x (n x 2) + 1) n-Stencil

Register Memory. The *Register Memory* is fast and efficient as it provides parallel/patronized data access in a single cycle to the accelerator. The register memory resides on the same hardware of processing unit and uses FPGA resources/slices that have multiple small independent memory register arrays. The HLS tools (such as ROCCC [11] & HCE [10]) can automatically solve memory bottleneck problems by splitting, interleaving, or reorganizing registers that reuse the portion of data. The HLS tools interleave and reorganize the register memory that rearrange sequential data storage into two or more non-contiguous storage blocks and provide single cycle access.

Specialized Memory. The application's dataset placed on main memory (SDRAM) has a linear structure. Hence, it is critical to keep the application's dataset access latency much lower than the main memory. The *Specialized Memory* (Figure 4 (b)) structure is used to reduce SDRAM access latency and provide parallel read-/write accesses to compute units. The *Specialized Memory* architecture is an array of dual port Static RAM cells which are tightly coupled and mapped according to the accelerator's access patterns and can be shared between more than one accelerators by the memory management unit 3.3. The *Specialized Memory* access is fast as register memory and is physically organized into multi dimension (1D/2D/3D) architecture. One application kernel can access complete row, column or block of *Specialized Memory*. The memory specified for one accelerator can be accessible to another accelerator if data reuse is required.

The program structure that is used to initializes *Specialized Memory* into 1D, 2D or 3D structure is shown in Figure 4 (a). The *Buffer_Width* describes the size of row *Nx* (1D buffer). The *Buffer_Dimension* explains the dimensions (2D/3D)

(*Nz and Ny*) of the memory block. *Nx* and *Nz* define the sizes of each block (Ny). The size of block (Nx, Nz) is selected to fit in one BRAM of the target device. The current AMC *Specialized Memory* architecture is composed of single or multiple BRAM18 (1x18 Kb) blocks. Depending upon the dimension of *Specialized Memory* (Nx, Nz) (rows, column) the memory block can be configured as, `1x16k, 2x8k, 4x4k, 8(9)x2k` or `16(18)x1k`. A *master port* (M) is used to transfer streaming data between main memory unit and *Specialized Memory*.

For example, a 3D *Specialized memory* architecture for generic stencil data access is shown in Figure 4 (b). Each bank of *Specialized memory* has two ports (PortA & PortB) which allows the compute unit to perform parallel read/write data to/from multiple banks. The on-chip encapsulation of data memories with processing units allows application kernels to access data without extra delay. The generic stencil structure is shown in Figure 4 (c). When n=4, 25 points are required to compute one central point. This means, to compute a single element eight points are required in each of x, y and z directions, as well as one central point. The 3D-stencil (when n = 4) occupies 9 different banks to place it on *Specialized Memory*. The central, x and z points are placed on the same bank(y), and each y-point needs a separate bank (y+4 to y-4). The *Specialized Memory* takes 9 cycles ((2 points/bank)/cycle) to transfer 17 points of bank(y) and 1 point each from bank(y+4) to bank(y-4). Each bank read/write operation is independent to other and can be performed in parallel. Therefore, the *Specialized Memory* takes nine cycles to transfer 25 points of single stencil. To utilize all bank and to reduce access time multiple 3D-Stencils are placed on 3D *Specialized Memory* (Figure 5 (b).

Global Memory. The slowest type of memory in the AMC system is the global memory, which is the main memory (SDRAM), and is accessible by the whole system. Even though, AMC has an efficient way of accessing main memory that best utilizes the bandwidth, it still has latency with respect to other memory systems (discussed in section 3.5).

3.3. Memory Manager

To achieve maximum memory bandwidth, the AMC organises complex access pattern in single or multiple descriptors at compile-time. At run-time, the AMC *Memory Manager* transfers complete pattern to/from the main memory and *Specialized Memory* in multiple noncontiguous strided streams. The AMC *Memory Manager* has a view of main memory address space that is partitioned into

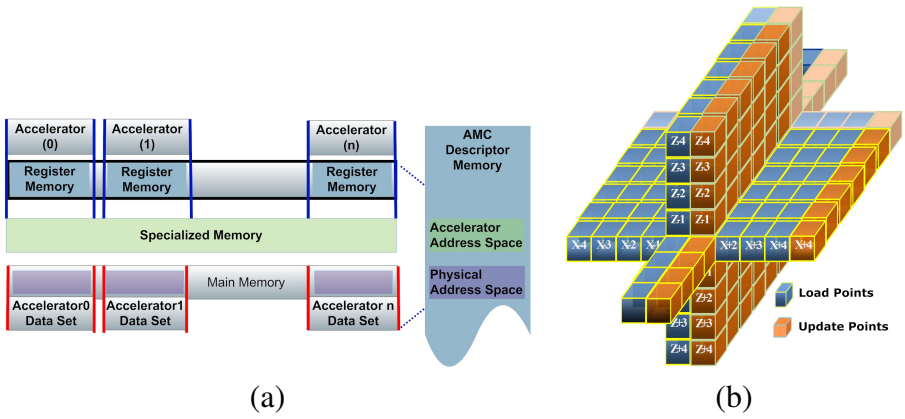


Figure 5: (a) Memory Hierarchy of AMC System (b) N-Stencil Vector Load & Update Points

segments (shown in Figure 5 (a)). Each segment contains single or multiple descriptors that hold the information of specialized and main memory for each accelerator. The AMC *Memory Manager* applies protection at the segment level e.g. a segment can be read/written by the accelerator for which it is allocated. Within a segment, AMC organizes and rearranges multiple noncontiguous memory accesses simultaneously that reduces read/write delay due to the control selection of SDRAM memory. To reduce false sharing, the *Specialized Memory* is by default dedicated to a single accelerator unit. In the AMC system, the *Specialized Memory* is tightly connected to the accelerator that avoids core stalling due to transfer of memory content in terms of on-chip communication and SDRAM access latency. AMC keeps the knowledge of memory as to whether or not a certain memory area is in the accelerator's *Specialized Memory*. This knowledge allows the AMC to manage the placement of memory as well as reuses and shares already accessed memory. The *Memory Manager* is further divided into two sections which are: the *Address Manager* and the *Data Manager*. The *Address Manager* fetches single or multiple descriptors depending on the access pattern, translates/reorders in hardware, in parallel with AMC Read/Write operations. The *Data Manager* improves the *(computed$_{point}$ / accessed$_{point}$) (c/a)* ratio by organizing and managing the memory accesses. For an accelerator generating single *computed$_{point}$*, the maximum achievable (ideal) *c/a* ratio is 1. To provide efficient data access and excessive reuse, the *Data Manager* is further divided into three units: the *Load Unit*, the *Update Unit* and the *Reuse Unit*. The *Load unit* accesses all point of an access pattern, which are required for single *Computed$_{point}$*. After accessing the first access pattern (*points*), the memory manager transfers control to *Reuse unit*

and *Update unit*. The *Reuse unit* keeps reusing input *points* as much as possible. The *Update Unit* is responsible to update remaining memory access (*points*) required for the application kernels. For example, a generic stencil structure (shown in Figure 5 (b)) when n=4, requires 25 points to compute one central point. This means the $computed_{point}/access_{points}$ *(c/a)* ratio is `0.04`. To improve the *(c/a)* ratio the *Data Manager* is deployed to increase data reuse by feeding data efficiently to the computation engine. The *Data Manager* accesses multiple stencils from the input volume in the form of a stencil vector. This is shown in Figure 5 (b). *(N × (1+(n× 4)) + (n × 2))* represents the size of a single 3D-Stencil vector. Here, N represents the number of planes. Since the 3D-stencil's output is generated by processing consecutive neighboring points in 3 dimensions, a large amount of data can be reused with an efficient data management. The *Load unit* accesses the 3D-Stencil vector at the start of each row of the 3D-Memory volume. For the following vector access, the Load unit transfers control to the Update unit. For a single 3D-Stencil volume, number of accessed points is dependent on the number of planes and the stencil size. For example, a 3D-Stencil memory architecture having 24 base planes and 25 points required to compute one stencil point. In this case, a single Stencil vector (Figure 4 (d)) needs `600` points. The load unit reduces these numbers by reusing the points in the y-dimension. These points ($point_y$) are reused when they are part of a neighboring plane of the stencil vector. The number of Load unit points is mentioned in Equation 1, where $Ghost\_point_z$ refers to the points of the extended base volume and $Point_c$ indicates the central point.

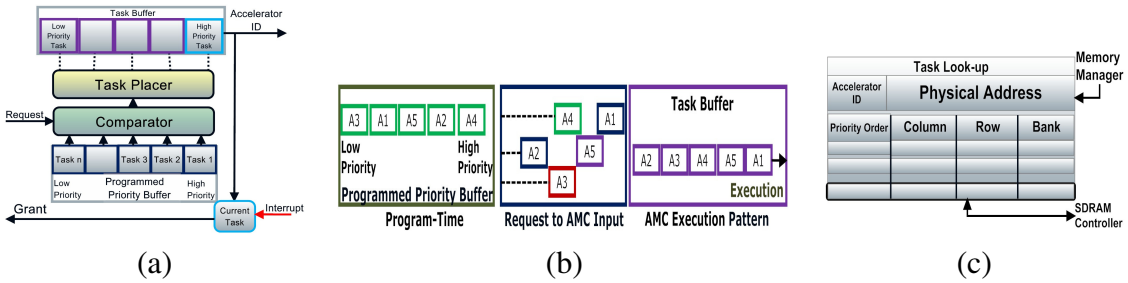
$$Eq = (Planes \times (Point_x + Point_z + Point_c)) + Ghost\_Point_z \quad (1)$$

After reading the first stencil vector, the *Memory Manager* shifts the control to the *Update Unit*. For each new access, the *Update unit* slides the stencil vector towards the x-direction and accesses further points while reusing neighboring points. The number of points required by the *Update Unit* for a stencil vector is presented in Equation 2.

$$Eq = (Planes \times (Point_x + Point_c)) + Ghost\_Points_z \quad (2)$$

After accessing the first row, the Reuse unit accesses the rest of the volume. This unit accesses only the central point and generates a stencil vector while reusing the existing rows and columns as mentioned in Equation 3.

$$Eq = (Planes \times Point_c) + Ghost\_Points_z \quad (3)$$



(a) (b) (c)

Figure 6: AMC: (a) Front-End Scheduler (b) Front-End Scheduling Process (c) Back-End Scheduler Lookup Table

The Memory Manager improves 3D-Stencil *(c/a)* ratio. Form the *8-point 3D-Stencil* unit, the Memory Manager uses a single input point 25 times before discarding it from the internal memory. In practice, this is not achievable due to the ghost points (i.e., points belonging to a neighboring tile that are necessary for the current computation) present on the boundaries of the input volume. The Memory Manager improves data reuse ratio for large base volumes.

3.4. Scheduler

The AMC scheduler manages read/write memory accesses and controls operations of multiple accelerators. The scheduler is categorized into two sections, the *Front-End Scheduler* and the *Back-End Scheduler*.

3.4.1. The Front-End Scheduler

The *front-end scheduler* deals with multi-accelerator execution processes. Each accelerator requests to access main memory, the *front-end scheduler* keep placing incoming requests in the buffer. The selection of an accelerator depends on the accelerator's request and priority state. The accelerators are categorized into three states, *busy* (accelerator is processing on local buffer), *requesting* (accelerator is free), and *request & busy*. In the *request & busy* state the accelerator is assumed to have double or multi buffers. During this state, the accelerator is processing on one buffer while making requests to fill other. To provide a feature of multi-buffers support in the current developed platform, a *state controller* (Figure 9 (b)) is instantiated with each accelerator that handles the states of accelerator using a double/multi-buffering technique. The *state controller* manages the accelerator's *Request & Grant* signals and communicates with the scheduler. Each *Request* includes a read and write buffer operation. Once the request is accepted, the state controller provides a path to the AMC read/write buffer.

AMC *front-end scheduler* supports two scheduling policies, *symmetric* and *asymmetric* that execute accelerators efficiently. In **Symmetric** multi-accelerator strategy, the AMC scheduler manipulates the available accelerator's request in FIFO (First in First out). The *Programmed Priority Buffer* (Figure 6 (a)) is used to manage the accelerator's request in FIFO order. The **Asymmetric** strategy emphasizes on priority and incoming requests of the accelerators. The scheduling policies are configured statically at program-time and are executed by hardware at run-time. The number of priority levels can be configured for asymmetric scheduling. Assigned priorities of the accelerators are placed in the *programmed priority buffer* (Figure 6 (a)). The *comparator* picks an accelerator to execute, only if it is ready to run, and there are no higher priority accelerators that are in a ready state. If same priorities are assigned for more than one accelerator, AMC scheduler executes them as FIFO method. If no scheduling policy is defined for the *front-end scheduler*, AMC transfers control to the *back-end scheduler*.

The *Front-End Scheduler* accumulates requests from the multi-accelerator system and maintains them in the *task buffer* (Figure 6 (a)) as per predefined scheduling policy. For example, different requests are generated in concurrent order from multiple sources (Shown in Figure 6 (b)). The *Front-End Scheduler* takes first request as it is, remaining requests are executed on the priority level defined in *Programmed Priority Buffer*. When currently running accelerator process finishes, the AMC does a context switch. Depending upon the scheduling policies and programme priority buffer, the AMC selects the new accelerator.

3.4.2. The Back-End Scheduler

The AMC *Back-End Scheduler* employs a strategy that gathers multiple memory requests, manages them with respect to physical addresses (SDRAM) and maximizes the reuse of open SDRAM banks that decrease the overhead of opening and closing of rows. This strategy imposes conditions on the arrangement of the memory accesses and affects the worst-case latency and gross/net bandwidth of external memory. The *back-end scheduler* uses two scheduling rules for memory patterns:

- Patterns are scheduled in a non-blocking manner, which means that a pattern that has been issued cannot be stopped until it has finished access.
- A read or a written pattern scheduled immediately after itself, when the memory is idle. This makes consecutive read and writes accesses independent of each other, further simplifying back-end scheduling operation.

The scheduling of memory accesses is dependent upon the physical address of current and next transfer. For example, at run-time the *Back-End Scheduler* gathers memory requests from multi-accelerator and place them in *Address Look-up Table* (shown in Figure 6 (c)). The *Address Look-up Table* contains unordered memory access requests. Each memory access is categorized into four parts i.e. *ID*, *Bank*, *Row* and *Column*. The *ID* holds the information of an accelerator and its local memory. The *Bank*, *Row* and *Column* belongs to the physical memory address space. At run-tome, the *Back-End Scheduler* schedules memory accesses of multiple accelerators by giving highest priority to the bank and row address. The lookup table executes the AMC policy called bank/row address management policy. The policy manages addresses in a lookup table so that it accesses SDRAM memory that is available in the row buffer or have the same bank without conflicting the pattern. Accessing memory from the same row buffer is the fastest memory access and only requires column access. The longest memory access (when Bank conflicts) requires a pre-charge signal, followed by row and column accesses.

The Memory Manager (Section 3.3) of AMC has particular descriptor memory (register set) for each accelerator unit, shown in Figure 5 (a). These descriptors are masked with interrupt and request signal. Once a request is generated the Memory Manager starts memory operation for the requested accelerator using its descriptors. After completion of memory read/write operation, the AMC scheduler receives an interrupt (*ack*) signal from the Memory Manager unit. This signal informs the scheduler about the selection of next accelerator to execute. The scheduler captures the *ack* signal from the Memory Manager and assigns the *grant* signal to the appropriate accelerator unit.

3.5. SDRAM Controller

In current AMC design, the SDRAM device uses four banks of memory per device therefore four address bits are used to select the memory bank. For the selection of the appropriate row and column within that row of memory 13 and 10 address lines are used respectively that completes the address mapping from physical address to the memory address. The memory controller has a peak bandwidth of 3.2 GB/s since it has a clock frequency of 200 MHz, a data rate of 2 words per clock cycle, and a data bus width of 64 bits. The memory controller [12] consumes 30 cycles for single load/store memory access. Out of which 1-3 clocks are used to communicate the memory controller with requesting unit. The PRECHARGE command is used to read/write row from a bank. To activate row of given bank, ACTIVATE command is inserted. Once the Read/Write

API Function	Description
`send (accelerator_id, stream, stride)` `receive (accelerator_id, stream, stride)`	*Vectorizing Data Access* Stride defines type of vector for the acclerator stride = 1 transfers a row vector stride = row, transfers a column vector stride = row + 1, transfer diagonal vector
`AMC_MEMCPY (command, accelerator_buffer, dataset)`	*Block/Tiled Data Access* command defines access type read or write `accelerator_buffer` indicate specialized memory buffer dataset indicates main memory block of data

Table 1: C/C++ Device Drivers to Program/Operate PPMC System

commands are given to SDRAM with credible data, it takes ten empty clocks to precharge using PRECHARGE command. The PRECHARGE to ACTIVE command takes four cycles and ACTIVE to READ command also takes four cycles. READ/WRITE command to read/write valid assertion takes ten cycles. Reads and writes occur in bursts.

4. Use Case Example

To explain the working principle of the AMC system, in this section we briefly describe its supported access patterns and programming.

The current AMC system provides comprehensive support for the C and C++ languages to control the HLS multi-accelerator system and data accesses (Vectorizing/Tiling). The functionality of the AMC is managed by C based device drivers shown in Table 1. The device driver aligns and arranges multiple data stream in single/multiple descriptor/s at compile time, this reduces on-chip communication and address request/grant time. The AMC programming interface allows the programmer to describe complex access patterns using C/C++ APIs and align data structures at program-time; during compilation AMC realigns application kernels that demand a selective layout of *Specialized Memory*. AMC device drivers allow the programmer to describe *Specialized Memory*. To manage a number of accelerators, each accelerator is assigned a fixed id. Data transfer is done using explicit event (message passing) between different accelerators controlled by AMC scheduler. If an accelerator needs data from on-chip or off-chip memory, it sends a request signal to AMC Front-End Scheduler which manages the data

movement. If zero priority is assigned to an accelerator (at compile-time) then AMC scheduler transfers control to *Back-End Scheduler*. The Back-End scheduler executes input request with respect to open SDRAM banks and rows. The highest priority is assigned to accelerate's addresses with open bank and row. An `AMC_MEMCPY` instruction is created which reads/writes a block of data from main memory to the accelerator's *Specialized Memory*.

Figure 7 shows the programming of AMC based multi-accelerator system. The program initializes two hardware accelerators and their 2D and 3D tiled data pattern. Part I & II of defines the specialized memory and data set. Part III of the program structure specifies the scheduling policies that include the accelerator id and its priority. Part lV shows `AMC_MEMCPY` instructions that copy 2D/3D blocks of data from the main memory to the specialized memory of accelerator. The AMC scheduler supports the scheduling policies similar to the Portable Operating System Interface (POSIX) Threads. If the same priority is assigned to multiple accelerators, the AMC scheduler processes the accelerators in symmetric mode.

```
// AMC: Specilized Memory
typedef struct DATA_MEMORY
    {
// On-Chip Specilized Memory Description
    int BUFF_WID; //  WIDTH
    int BUFF_DIM; //  DIMENSION
    }AMC_Specilized_Memory;

// AMC: DataSet
typedef struct DATASET_MEMORY
    {
// On-Chip Specilized Memory Description
    int DataSet_ADD; // BASE ADDRESS
    int DataSet_WID; // WIDTH
    int DataSet_DIM; // DIMENSION
    }AMC_DataSet;

typedef struct Accelerator
    {
// Accelerator Description
    int Accelerator;
    int Priority;
    }AMC_Acclerator;

  int main(){
    AMC_Specilized_Memory Buff_2D;
    AMC_Specilized_Memory Buff_3D;
    AMC_Acclerator Accelerator0;
    AMC_Acclerator Accelerator1;
    AMC_DataSet DataSet0;
    AMC_DataSet DataSet1;
```

```
// Part I
  // Initilizes 2D (128x128) Specilized Memory
    Buff_2D.BUFF_WID=128;
    Buff_2D.BUFF_DIM=2;
  // Initilizes 3D (32x32x32) Specilized Memory
    Buff_3D.BUFF_WID=32;
    Buff_3D.BUFF_DIM=3;

// Part II
  // DATA SET : Golab Memory
    DataSet0.DataSet_ADD=0X00000000;
    DataSet0.DataSet_WID=128;
    DataSet0.DataSet_DIM=2;
    DataSet1.DataSet_ADD=0X00010000;
    DataSet1.DataSet_WID=32;
    DataSet1.DataSet_DIM=32;

//Part III
  // Accelerator Initilization
    Accelerator0.Accelerator=0;
    Accelerator0.Priority=1;
    Accelerator1.Accelerator=1;
    Accelerator1.Priority=2;

//Part IV
  // DataTransfer
    AMC_MEMCPY{Accelerator0,
    Buff_2D,
    DataSet0);
    AMC_MEMCPY{Accelerator1,Buff_3D,
    DataSet1); }
```

Figure 7: AMC Multi-Accelerator System: Programming Example

5. Experimental Framework

In this section, we describe and evaluate the AMC based HLS multi-accelerator system. In order to evaluate the performance of the AMC system, the results are compared with a MicroBlaze and Intel core based HLS multi-accelerator systems. The Xilinx Integrated Software Environment and Xilinx Platform Studio are used to design the HLS multi-accelerator system. Xilinx Power Estimator does the power analysis. A Xilinx Virtex-5 ML505 evaluation FPGA board is used to test the multi-accelerator systems. The section is divided into four sub-sections the *Intel based HLS Multi-Accelerator System*, the *MicroBlaze based HLS Multi-Accelerator System*, the *AMC based HLS Multi-Accelerator System* and the *HLS Multi-Accelerator Kernels*.

5.1. Intel based HLS Multi-Accelerator System

The Intel system with HLS multi-accelerator is shown in Figure 8 (a). The system architecture is further divided into three sections the *Master Core*, the *Programming API*, and the *Bus Unit*.

5.1.1. The Master Core

The Intel Core i7 CPU is used to manage the memory system and schedules the HLS multi-accelerator. To achieve the maximum performance the HLS multi-accelerator system is executed on an optimized multi-threaded reference implementation, written in C++, compiled with g++ with optimization -O3, and executed on system having a quad-core Intel Core i7-2600, 3.4 GHz, with 16 GB RAM, 1333 MHz bus. The system uses Ubuntu 11.04 OS with Linux kernel version 2.6.3. These higher memory baselines are required to enable sufficient memory for the HLS multi-accelerator kernels. In the current system, Performance Application Programming Interface (PAPI) hardware performance counter used to collect execution clock cycles for each accelerator kernel.

5.1.2. The Programming API

OpenMP (Open Multiprocessing) is an API that caters multi-platform shared memory systems and extend it beyond real HPC systems that contain embedded systems, real-time systems, and accelerators. Tasks execution is the most significant feature of OpenMP 3.0. OpenMP 3.0 task pragmas make it compatible with the idea of using one HLS multi-accelerator. The shared memory parallelism is specified by C/C++ program using a set of compiler directives, runtime routines and environment variables that power the run-time performance of the system. In

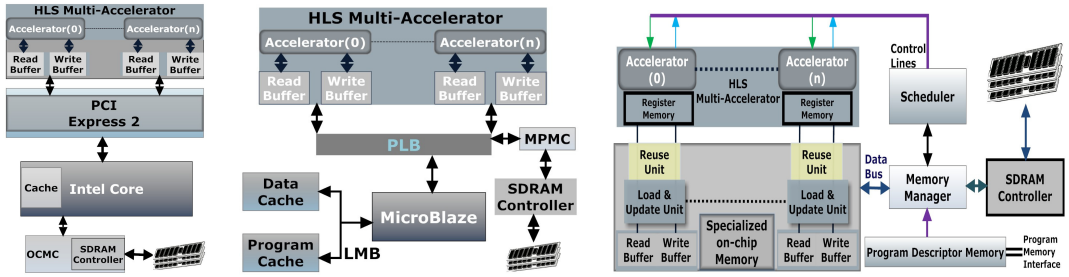


Figure 8: HLS Multi-Accelerator Systems: (a) Intel Core System (b) MicroBlaze System (c) AMC System

current multi-accelerator system, each accelerator is defined in a separate task. OpenMP executes them independently ensures that all defined accelerators and data transfer tasks are completed at some point.

5.1.3. The PCI Bus Unit

Intel core manages data movement between *On Chip Memory Controller* (OCMC) and multi-accelerator by using PCI bus. The PCI bus has multiple DMA channels which manages the data transfer requests. The data transfer requests issued to PCI DMA channels are according to available completion resources (Send/Receive interfaces) which forward them to the appropriate DMA channel. The XpressLite2 IP [13] is used in the design to manage eight separate data flows using DMA channels. The PCI Express XpressLite2 IP transfers the data sets from the host machine to multi-accelerator. The PCI Express IP is programmed to work at 1 G Byte/s using a 125 MHz clock speed and a 64-bit data bus.

5.2. MicroBlaze based HLS Multi-Accelerator System

The FPGA based MicroBlaze system is proposed (Figure 8 (b)) to execute HLS multi-accelerator kernels. The design (excluding hardware accelerators) uses 7225 flip-flops, 6142 LUTs and 15 BRAMs. The system architecture is further divided into four sections the *Master Core*, the *Bus Unit*, the *Memory Unit* and the *Programming API*.

5.2.1. The Master Core

A MicroBlaze softcore is used in the HLS multi-accelerator system that schedules the memory requests of HLS multi-accelerator kernels. The MicroBlaze processor [14] has Harvard memory architecture where instruction and data accesses

have separate 32-bit each address spaces. MicroBlaze instruction prefetcher improves the system performance by using the instruction prefetch buffer and instruction cache streams. The accelerators scheduling and data memory management are controlled by MicroBlaze (RISC) soft-core processor. The data access to Input/Output memory is memory mapped. The MicroBlaze system has two interfaces for memory and I/O accesses; the Local Memory Bus (LMB), the Processor Local Bus (PLB). The MicroBlaze uses a dedicated Local Memory Bus (LMB) [15] to link with local-memory (FPGA BRAM) that offers single clock cycle access to the local BRAM.

5.2.2. The Bus Unit

In the design, a Processor Local Bus (PLB) [16] provides connection between hardware accelerators and microprocessor. The PLB has 128 bit-width and connected to a bus control unit, a watchdog timer, separate address/read/write data path units, and an optional DCR (Device Control Register) slave interface that provides access to a bus error status registers. Bus is configured for single masters (MicroBlaze) and multi slaves (HLS multi-accelerators). An arbiter is used that grants the data access to multi-accelerator. The Input/Output (I/O) Module [17] is a light-weight implementation of a set of standard *(*I/O) functions commonly used in a MicroBlaze processor sub-system. The *(*I/O) bus provides access to external modules using MicroBlaze Load/Store instructions. The Input/Output Bus is mapped in the MicroBlaze memory space, with the *I/O* bus address directly reflecting the byte address used by MicroBlaze Load/Store instructions. The PLB provides maximum of 2 GByte of bandwidth while operating at 125MHz and 128-bit width, with byte enables to write byte and half-word data.

5.2.3. The Memory Unit

Local memory is used for data and program storage and is implemented using Block RAM. The local memory is connected to MicroBlaze through the Local Memory Bus (LMB) and the LMB BRAM Interface Controllers. The target architecture has 16 KB of each instruction and data cache. To access data from main memory, a parameterizable Multi-Port Memory Controller (*MPMC*) [18] is employed. *(MPMC)* provides an efficient interfacing between the processor and SDRAM. The *MPMC* connects SDRAM with MicroBlaze processors using IBM CoreConnect Processor Local Bus (PLB). A DDR2 controller is used with *MPMC* to access data from DDR2 SDRAM memory. The supported DDR2 memory has a peak bandwidth of 1 GByte/s as it has a clock frequency of 125MHz, and a data bus width of 64 bits.

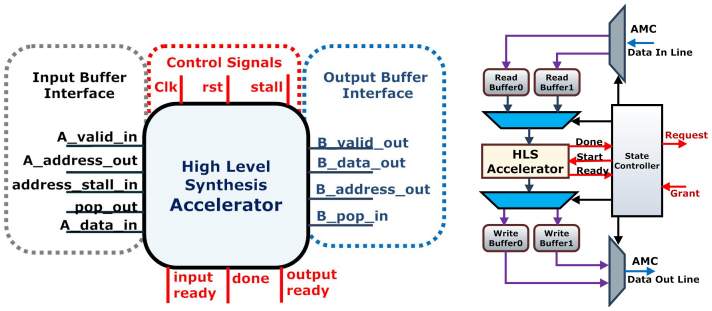
5.2.4. *The Programming API*

A small light-weight easy to use Real-Time Operating System (RTOS) Xilkernel [19] is experienced on the MicroBlaze soft processor. Xilkernel is highly integrated into the design tools of Xilinx that makes it possible to configure and build an embedded system using MicroBlaze and Xilkernel. Xilkernel API performs scheduling, inter-process communication and synchronization with a Portable Operating System Interface *POSIX* interface. The Xilkernel POSIX support statically declares threads that start with the kernel. From the main, application is spawn into multiple parallel threads using pthread library. Each thread controls a single HLS accelerator and its memory access. The software application consists of Xilkernel and application kernel threads executing on top of the main program.

MicroBlaze system uses Xilinx Software Development Kit (SDK) [20] that compiles the application kernels using library generator (libgen) [21] and a platform-specific gcc/g++ compiler and generate the final executable object file. The Xilinx library generator (libgen) is used to produce libraries and header files necessary to build an executable file that controls HLS multi-accelerator. Libgen parses the system hardware and sets up drivers, interrupt handling, etc. and creates libraries for the system. The libraries are then used by the MicroBlaze GCC compiler to link the program code for the MicroBlaze based HLS multi-accelerator system. The object files from the application and the Software Platform are linked together to generate the final executable object file.

5.3. *AMC based HLS Multi-Accelerator System*

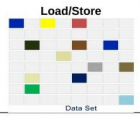
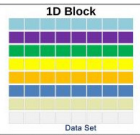
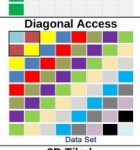
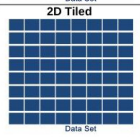
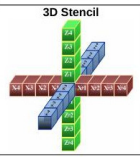
The AMC system is shown in Figure 6 (c). AMC controls the HLS Multi-Accelerator and performs scheduling and memory management without intervention of microprocessor or operating system (OS). In the current implementation of



(a) (b)

Figure 9: HLS: (a) Hardware Accelerator (b) State Controller

Table 2: Brief description of application kernels

Application Kernel	Description	Access Pattern	*Reg, LUT*	GFLOPS
Radian Converter	Converts Degree into radian		68, 67	0.375
Thresholding	An application of image segmentation, which takes streaming 8-bit pixel data and generates binary output.		2289, 2339	0.125
FIR Finite Impulse Response	Calculates the weighted sum of the current and past inputs.		3953, 2960	3.875
FFT Fast Fourier Transform	Used for transferring a time-domain signal into corresponding frequency-domain signal.		4977, 2567	6.0
Matrix Multiplication	Output= Row[Vector] × Column[Vector] X=Y×Z		2925, 1719	7.750
Smith Waterman	Determining optimal local alignments between nucleotide or protein sequences		3380, 2616	1.125
Laplacian Solver	Applies discrete convolution filter that can approximate the second order derivatives.		6977, 5567	2.125
3D-Stencil Kernel	An algorithm that averages nearest neighbor points (size 8x9x8) in 3D.		6205, 2853	4.625

AMC, a modular DDR2 SDRAM [22] controller is integrated that accesses data from physical memory and to perform the address mapping from physical address to the memory address. The DDR2 SDRAM controller provides a high-speed source-synchronous interface and transfers data on both edges of the clock cycle. The DDR2 memory has a peak bandwidth of 1 G Byte/s since it has a clock frequency of 125MHz, a data rate of 2 words per clock cycle, and a data bus width of 32 bits. A 256 MByte (32M x 16) of DDR2 memory having SODIMM I/O module is connected with AMC memory controller. The system (excluding HLS accelerators units) consumes 4986 flip-flops, 4030 LUTs, 24 BRAMs.

5.4. HLS Multi-Accelerator Kernels

The application kernels that are used in the design are shown in Table 2. The hardware accelerator (shown in 9 (a)) is generated by the HLS ROCCC[11] compiler for the evaluated application kernel. A wrapper module and a *state controller* (Figure 9 (b)) is integrated with each hardware accelerator to manage multiple buffers and makes it feasible to be integrated in the AMC, MicroBlaze and Intel core based systems. Column *Access pattern* of Table 2 presents memory access patterns of the application kernels. Each color represents a septate memory access pattern. The column *Reg, LUT* describes the slices (Registers, LUTs) utilized by HLS-based hardware accelerators. The column *Points* shows a number of inputs $(access_{point})$ required to generate a single output.

6. Results and Discussion

This section analyzes the results of different experiments conducted on AMC, MicroBlaze and Intel based systems. The experiments are characterized into four subsections: *Application's Performance*, *System's Performance*, *Memory Access Unit*, and *Area & Power*.

6.1. Application's Performance

The application kernels (Table 2) are executed individually on AMC, MicroBlaze and Intel based systems. The section is further divided into two subsections that are: *Applications Performance without On-Chip Memory* and *Applications Performance with On-Chip Memory*.

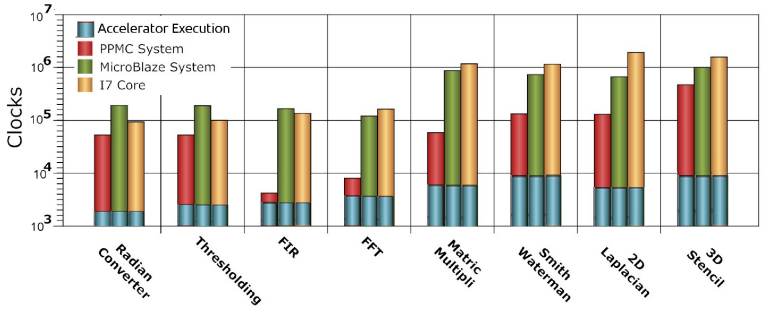


Figure 10: Application Kernels Execution Time Without On-chip Memory Support

6.1.1. Applications Performance without On-Chip Memory

This section presents execution time (clock cycles) of each application kernel by disabling on-chip (specialized/cache) memories of the systems (shown in Figure 10). Each bar represents the application kernel's computation time and memory access time. The application kernel computation time contains the HLS accelerator processing time for 4KByte of data. Memory access time holds address/data management and request/grant time from main memory unit. X and Y axis represent application kernels and number of clock cycles, respectively. The vertical axis has logarithmic scale in the Figure 10. By using AMC system, the results show that Radian converter achieves 3.94x and 1.75x of speed-up compared to the MicroBlaze and Intel based systems respectively. The Thresholding application achieves 7.1x and 1.89x of speed-up. These applications have load/store memory access pattern. The FIR application has streaming data access pattern with 43.2x and 32x of speed-up. The AMC system requires only one descriptor block to access data pattern thus reduces address generation/management and on-chip communication time. The FFT application kernel reads a 1D block of data, processes it and writes it back to physical memory. This application achieves 22x and 20x speed-ups. The Matrix Multiplication kernel accesses row and column vectors. The AMC system manages complex data patterns of the application in hardware and attains 29x and 19.9x of speed-up. The Smith-Waterman application achieves 10.4x and 8.7x of speed-up. The Laplacian filter achieves 6.15 and 14.83 of speed-up. Both Laplacian and Smith-Waterman applications have 2D Tiled (block) access pattern. The 3D-Stencil data decomposition achieves 3.6x and 3.3 of speed-up.

6.1.2. Applications Performance with On-Chip Memory

In this section, we calculate $computed_{point}/access_{point}$ (c/a) ratio of AMC system for each application kernel and compared the performance with MicroBlaze and Intel systems. The column *c/a without memory manager* of Table 6.1.1 represents data elements required to generate a single output. *Radian Convertor & Thresholding* kernels has load/store access pattern with one *c/a* ratio, it means only a single element from main memory is required to the computing unit. Due to irregular data access pattern both applications are unable to get the benefit from on-chip memory. As shown in Table 6.1.1 & Figure 11 the application *FIR, FFT, Matrix Multiplication, Smith-Waterman, 2D Laplacian, and 3D-Stencil* need more than a single data elements for next computations. A 128-Tap FIR filter requires 128 number of inputs to generate a single output. The AMC Data Manager improves c/a ratio of FIR and 2D Laplacian Kernel to one by reusing accessed points.

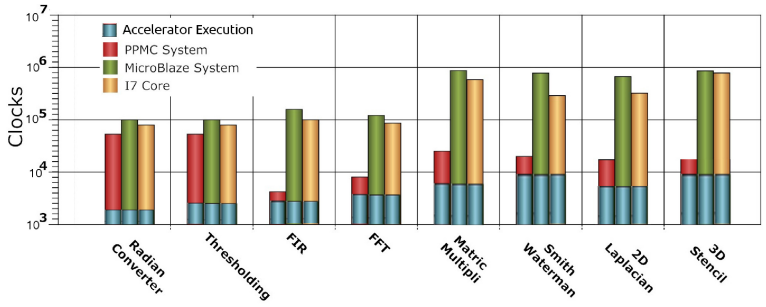


Figure 11: Application Kernels Execution Time with On-chip Memory Support

FFT and Smith-Waterman kernel's data elements are not reused by AMC system due to complex access pattern. Matrix Multiplication application kernel takes 32 element wide row and column vector to generate a single element. The AMC system reused row vector and accesses only column vector for each multiplication. For generic stencil (n=4) application kernel, 25 points are required to compute one central point. This means the $computed_{point}/access_{point}$ *(c/a)* ratio is `0.04`. The AMC memory system improves the *(c/a)* ratio by reusing and feeding data efficiently to the computation engine. The AMC Memory system keeps updating/using all memories, so that memory accesses do not affect the performance of the system. When executing application on the system by enabling on-chip

Table 3: AMC: On-Chip Memory System

Application Kernel	Load Value	Reuse Value	Update Value	*c/a* without Memory Manager	Achieved *c/a* with Memory Manager
Radian Convertor	1	0	1	1	1
Thresholding	1	0	1	1	1
FIR	128	127	1	0.0078	1
FFT	64	0	64	0.015	0.015
Matrix Multiplication 32x32 (Row x Column)	64	32	32	0.015	0.031
Smith Waterman	3	0	3	0.33	0.33
2D Laplacian (3x3)	9	8	1	0.1	1
3D Stencil (8x9x8)	25	24	1	0.04	0.125

(cache/specialized) memory units results show that Radian converter and Thresholding applications do not improve performance due to their irregular memory pattern and having no temporal data locality. While executing FIR application, the results show that AMC system achieves 26.5x and 14.4x of speed-up compared to MicroBlaze and Intel systems respectively. The FIR application has streaming data access pattern with maximum sequential data locality. The FFT application kernel reads a 1D block of data, processes it and writes it back to the main memory. This application has achieves 11x and 8.5x speed-ups. The Matrix Multiplication kernel accesses row and column vectors. The application attains 14x and 9.6x of speed-up. The Smith-Waterman and Laplacian application have 2D block/tiled data access pattern. The Smith-Waterman application has no data locality and achieves 36.3x and 14.3x of speed-up. The Laplacian filter has temporal data locality and achieves 38.8 and 18.73 of speed-up. The 3D-Stencil application kernel has 3D tiled memory access pattern with complex data locality. The AMC system's 3D data and descriptor memory manages/reuses 3D-Stencil data and 3D tiled access pattern respectively and achieves 58x and 53.7 of speed-up.

6.2. System's Performance

The system performance is measured by executing HLS multi-accelerator kernels all together on AMC, MicroBlaze and Intel based systems. All application kernels are executed simultaneously with the different set of priorities. At run time, AMC system and baseline systems manage executions and pipeline/overlap data transfer where possible. Figure 12 illustrates the execution time of the system and categorizes execution time into three factors: computation (application processing) time, arbitration (request/grant) time among the scheduling, and the memory management (bus delay and memory access) time. The computation time of application kernels in all systems is overlapped under the scheduling, and memory access time (shown in Figure 10), therefore, it is not shown in Figure 12. The Intel-based system holds PCI bus communication which takes extra time to access data from the *Main Memory*. In the AMC system, memory management time is dominant, and the AMC overlaps scheduling and computation under memory access time. While running all HLS accelerator kernels together, the results show that the AMC based system achieves 10.4x and 7x of speed-up compared to MicroBlaze and Intel Core based systems. The AMC system efficiently schedules HLS multi-accelerator and manages memory access patterns.

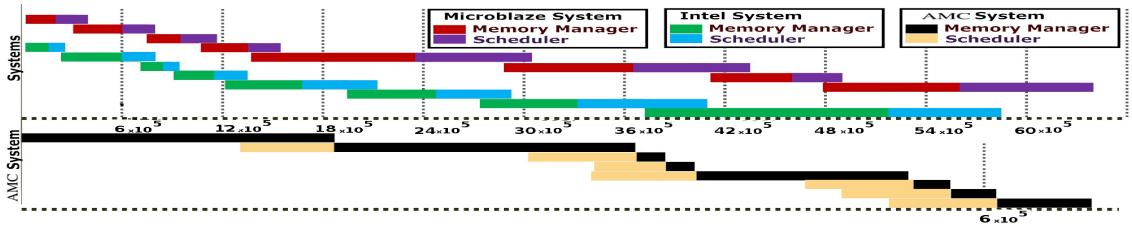


Figure 12: HLS multi-accelerator Systems Execution Time

6.3. Memory Access Unit

In this section we compare AMC system's data access time with Vector and MicroBlaze based FPGA systems, Intel Core i7 CPU and GPU Nvidia C2050 Fermi-based simulation environments by executing the *image thresholding* application with irregular memory access pattern. Figure 13 shows a plot with Read-/Write data accesses for AMC, MicroBlaze, Pentium, Vector and GPU core systems. The X-axis presents data sets that are read/written by the image thresholding application from/to the main memory. The Y-axis has logarithmic scale and presents the number of clock cycles consumed while accessing the data set. The main memory single access latency on a MicroBlaze system with 125MHz DDR SDRAM is measured to be almost 50 cycles. The SDRAM memory latency on Intel Core i7 CPU with a memory clock 1.33 GHz, SDRAM DDR3 Controller with capacity of 2x4 GB, SODIMM module and 128 bit bus-width is measured almost 150 cycles using SiSoftware Sandra 2013 [23]. While executing on FPGA, the VESPA vector processor core [24] consumes almost 300 clock cycles for a single load/store memory access. Streaming Architectural Simulator (SArcs) trace-based architectural simulator targeting real GPU device (Tesla C2050) based on the NVIDIA Fermi generation is used to evaluate GPU systems. A GPU Fermi architecture (C2050) having 3 GB SDRAM capacity and 144 GB/s bandwidth gets a penalty of approximately 600 clock cycles [25, 26] while accessing main memory. The results show that AMC irregular memory access is 2, 14, 26 and 20 times faster than MPMC, Intel core i7, vector and GPU systems respectively. The AMC system has higher speed-up for regular access patterns (such as Tiled) than for irregular patterns.

6.4. Area & Power

Studies [27] have shown that on a GPU system, applications are easily implemented and processed significantly faster than on FPGA and multi-core systems.

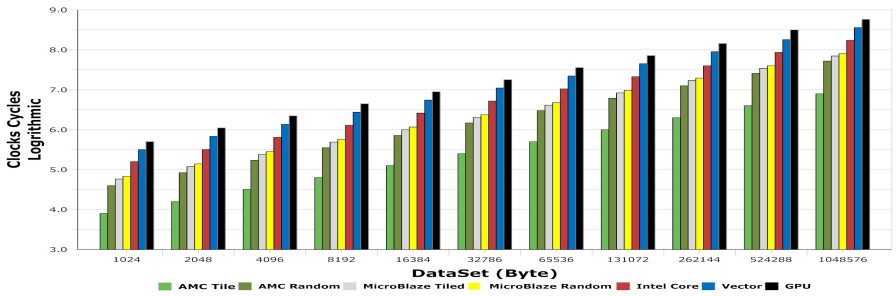


Figure 13: Memory Read/Write Access Time of Different Systems

However, a compute-capable discrete GPU can draw more than 200 watts [27] by itself. In this section, we measure the idle power of AMC, MicroBlaze and Intel systems without having HLS multi-accelerator system. The Intel Core i7 CPU (4 Cores) with a system clock of 2.4 GHz and 8 GByte of global memory consumes 15.80 watts static power [23]. The MicroBlaze and AMC systems without having HLS multi-accelerator consume 2.75 and 2.1 watts respectively on Xilinx V5-Lx110T FPGA device. Both systems have a system clock of 125 MHz and 256 MByte of global memory. Due to the light weight of AMC, the system consumes 32% fewer slices and 23% less on-chip power than the MicroBlaze based system.

7. Related Work

A programming model is significant for multi-core architectures as it automates the exploitation of parallelism of a sequential software and make it portable between different architectures. The Cell Superscalar (CellSs) [28] programming model permits programmers to write sequential application and the programming model utilizes the offered concurrency and uses the dissimilar components of the Cell/BE (PPE and SPEs) for automatic parallelization at run time. CellSs framework manages data transfer using task interface. The main memory is accessed by DMA routines. A DMA transfer or a group of DMA transfers is recognized by a tag. Number of transfers can be grouped by using the similar tag. After starting an asynchronous transfer, the completion can be assured via the DMA tag. An interface is provided to scatter/gather memory access patterns for one, two and three-dimensional arrays. Using the combination of different DMAs (DMA list) CellSs implements Scatter/Gather operations. Intel brings a suite of programming

models, the Intel Array Building Blocks [29].

A number of scheduling and memory management approaches exist for multi-accelerator, but to the best of our knowledge a challenge is still there to find mechanisms that can schedule dynamic operations while taking both the processing and memory requirements into account. Marchand et al. [30] developed software and hardware implementations of the Priority Ceiling Protocol that control the multiple-unit resources in a uniprocessor environment. A multi-accelerator having 16 AMD dual-core CPU computing nodes with 4 NVIDIA GPUS and a Xilinx FPGA is presented by Showerman et al. [31]. The cluster nodes are interconnected with both InfiniBand and Ethernet networks. The software stack consists of standard cluster tools, the accelerator-specific software package and enhancement of the resource allocation and branch subsystem. Ferrante [32] developed a scheduling algorithm which allows for distributing IPSec – a suite of protocols that provides security to communications at IP level – packet processing over the CPU and multiple accelerators and to support soft QoS. He provides some high-level simulations to prove that the algorithm works as desired and that it can provide a performance enhancement especially when the system is overloaded. Anderson [33] proposed and implemented a scheduling method for real-time systems for multicore platforms that encourage certain groups of tasks to be scheduled together while ensuring real-time constraints. This method can be applied to encourage tasks that shares a common working set to be executed in parallel, which makes more effective use of shared caches. Wolf et al. [34] Provides a real-time capable thread scheduling interface to the two-level hardware scheduler of MERASA multi-core processor. A time-bounded synchronization mechanism for the concurrent threads execution is proposed for multi-core architecture. The architecture is capable of executing hard real-time threads. Yan et al. [35] designed a hardware scheduler to assist the SPC task scheduling on heterogeneous multi-core architecture. The scheduler supports first come first service (FCFS) and dynamic priority scheduling strategies. Ganusov et al. [36] proposed the Efficient Emulation of Hardware Prefetchers via Event Driven Helper Threading (EDHT). EDHT gives the idea of using accessible general purpose cores in a chip multiprocessor environment. It acts as helper engine for separate threads working on the active cores. Wen et al. [37] present an FT64 based on chip memory subsystem that combines software/hardware managed memory structure. The stream accelerator is HPC application-specific coprocessor. It combines caching and software-managed memory structures, capturing locality exhibited in regular/irregular stream access without data transfer between stream register file and caches. Chai et al. [38] presented a configurable stream unit for providing stream-

ing data to hardware accelerators. The stream unit is situated in the system bus, and it prefetches and align data based on streams descriptors.

Hussain et al. [39] [40] discussed the architecture of a pattern based memory controller for application specific single accelerator. He also provides a memory controller [41] and [42] [43] for single core vector processor and graphics system respectively. The design is appropriate only for single core, whereas in AMC we present a mechanism both for multi-hardware accelerator. Moreover, features of AMC like the *Scheduler* and *Memory Manager* enable higher performance of HLS multi-hardware accelerators.

8. Conclusion

HLS-based multi-accelerator systems suffer from poor performance on FPGA architectures due to the processor-memory speed gap. A generic HLS multi-accelerator system requires a microprocessor (CPU) that controls the multi-accelerator and manages the memory system. In this work, we have proposed an efficient and intelligent controller in hardware (AMC) for a HLS multi-accelerator environment. The AMC improves the system performance by reducing accelerator/processor and memory speed gap, and schedule/manage complex memory patterns without the support of the processor and operating system. The AMC system provides strided, scatter/gather and tiled memory access support that eliminates the overhead of arranging and gathering address/data by the master core (microprocessor). The proposed environment can be programmed by the microprocessor using a High Level Language (HLL) API or directly from an accelerator using a specific command interface. The experimental evaluations based on the MicroBlaze and Intel-based HLS multi-accelerator systems with Xilkernel (RTOS) and Linux kernel respectively demonstrates that AMC system best utilizes hardware resources and efficiently accesses physical data. In the future, we are intending to insert a selective static/dynamic set of data access pattern inside AMC for multi-accelerator (vector accelerator) design that would definitely reduce the requirement of programming AMC by the user for a range of applications.

9. Acknowledgments

This work has been supported by the Ministry of Science and Innovation of Spain (CICYT) under contract TIN–2007–60625 and by the European Union Framework Program 7 HiPEAC2 Network of Excellence. The authors would like to thank the Barcelona Supercomputing Center, Microsoft's Barcelona Research

Center, Unal Center of Education Research and Development (UCERD) and the Universitat Politecnica de Catalunya (UPC) for their support. The authors also wish to thank the reviewers for their insightful comments.

References

[1] Xilinx Virtex-7. Leading FPGA System Performance and Capacity, 2012.

[2] Altera Stratix IV. Stratix IV FPGA: High Density, High Performance AND Low Power, 2012.

[3] Xilinx Artix-7. Leading System Performance per Watt for Cost Sensitive Applications, 2012.

[4] Halfhill, T.R. Tabula's time machine. *Microprocessor Report*, 2010.

[5] Kirk Saban. Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency. In *White Paper: Virtex-7 FPGAs*, 2011.

[6] Wulf, Wm. A. and McKee, Sally A. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 1995.

[7] McKee, Sally A. Reflections on the memory wall. *ACM: Proceedings of the 1st conference on Computing frontiers, 2004*.

[8] T. Hussain, M. Pericas, N. Nacho and E. Ayguade. Implementation of a Reverse Time Migration Kernel using the HCE High Level Synthesis Tool. *FPT 2012*.

[9] M.C.McFarland, A.C.Parker, and R.Camposano. The high-level synthesis of digital systems. *Proc. IEEE, vol. 78, no. 2*, 1990.

[10] Alessandro Marongiu and Paolo Palazzari. The HARWEST Compiling Environment: Accessing the FPGA World through ANSI-C Programs. *CUG 2008 Proceedings*, 2008.

[11] Villarreal, Jason, and others. Designing modular hardware accelerators in C with ROCCC 2.0. In *FCCM 2010*.

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

[13] PLDA. *PCI Express XpressLite2 Reference Manual*, February 2010.

[14] Embedded Development Kit EDK 10.1i. *MicroBlaze Processor Reference Guide*.

[15] Xilinx LogiCORE IP. *Local Memory Bus (LMB)*, December, 2009.

[16] Embedded Development KitEDK 10.1i. *MicroBlaze Processor Reference Guide*.

[17] Xilinx. *LogiCORE IP I/O Module*, October, 2012.

[18] Xilinx LogiCORE IP. *Multi-Port Memory Controller (MPMC)*, March 2011.

[19] Xilinx . Xilkernel, December , 2006.

[20] Xilinx Software Development Kit (SDK).

[21] Embedded System Tools Reference Manual EDK 13.1.

[22] Xilinx LogiCORE IP. *Multi-Port Memory Controller (MPMC)*, March 2011.

[23] SiSoftware Sandra 2013.

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

[25] D. Haugen. Seismic data compression and gpu memory latency. *Dept. of Computer and Information. Science, Norwegian University of Science and Technology, 2009.*

[26] Gulati, Kanupriya and Khatri, Sunil P. *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*. Springer Publishing Company, Incorporated, 2010.

[27] Scogland, and others. A first look at integrated GPUs for green high-performance computing. *Computer Science - Research and Development*.

[28] Cell Superscalar (CellSs) Users Manual (Barcelona Supercomputing Center), May 2009.

[29] A. Ghuloum, A. Sharp, N. Clemons, S. Du Toit, R. Malladi, M. Gangadhar, M. McCool. Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures, Sept, 2010 .

[30] P. Marchand and P. Sinha. A hardware accelerator for controlling access to multiple-unit resources in safety/time-critical systems. Inderscience Publishers, April 2007.

[31] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington and W. Hwu . QP: A Heterogeneous Multi-Accelerator Cluster. In *10th LCI International Conference on High-Performance Clustered Computing*, March 2009.

[32] A. Ferrante, V. Piuri and F. Castanier. A QoS-enabled packet scheduling algorithm for IPSec multi-accelerator based systems. In *Proceedings of the 2nd conference on Computing frontiers*, 2005.

[33] James H. Anderson and John M. Calandrino. Parallel task scheduling on multicore platforms. ACM, January 2006.

[34] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé", P. Sainrat, T. Ungerer. RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor. In *Proceedings of the 2010 13th IEEE International Symposium*.

[35] L. Yan, W. Hu, T. Chen, Z. Huang. Hardware Assistant Scheduling for Synergistic Core Tasks on Embedded Heterogeneous Multi-core System. In *Journal of Information & Computational Science (2008)*.

[36] Ganusov, Ilya and Burtscher, Martin. Efficient emulation of hardware prefetchers via event-driven helper threading. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 2006.

[37] M Wen, N Wu, C Zhang, Q Yang, J Ren, Y He, W Wu, J Chai, M Guan, C Xun. On-Chip Memory System Optimization Design for the FT64 Scientific Stream Accelerator. *Micro IEEE 2008*.

[38] Sek M. Chai, N. Bellas, M. Dwyer and D. Linzmeier. Stream Memory Subsystem in Reconfigurable Platforms. 2nd Workshop on Architecture Research using FPGA Platforms, 2006.

[39] Tassadaq Hussain, M. Pericas, N. Nacho and E. Ayguade. Reconfigurable Memory Controller with Programmable Pattern Support. *HiPEAC Workshop on Reconfigurable Computing*, Jan, 2011.

[40] Tassadaq Hussain, M. Shafiq, M. Pericas, N. Nacho and E. Ayguade. PPMC: A Programmable Pattern based Memory Controller. In *ARC 2012*.

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

[42] Tassadaq Hussain, Oscar Palomar, Adriyn Cristal, Osman Unsal, Eduard Ayguady, Mateo Valero and Amna Haider. Stand-alone Memory Controller for Graphics System. In *The 10th International Symposium on Applied Reconfigurable Computing (ARC 2014)*. ACM, 2014.

[43] Tassadaq Hussain and Amna Haider. PGC: A Pattern-Based Graphics Controller. *Int. J. Circuits and Architecture Design*, 2014.