CubeDMA - Optimizing Three-Dimensional DMA transfers for Hyperspectral Imaging Applications

Johan Fjeldtvedt, Milica Orlandić*

Department of Electronic Systems
Norwegian University of Science and Technology - NTNU
Trondheim, Norway

Abstract

Onboard computing is one of the principal needs in space-related technology in the recent years. In particular, onboard hyperspectral imaging (HSI) processing has advanced significantly. Due to advances in sensor technology, onboard HSI processing continuously meets new challenges related to increasing dataset size, limited processing time and limited communication links. High throughput and data reduction are crucial for satisfying real-time constraint and for preserving transmission bandwidth. For systems capable of accommodating a wide range of processing algorithms, there is a need for a flexible communication infrastructure that can provide fast access to/from memory in different access patterns. In this paper, existing FPGA-related Direct Memory Access (DMA) solutions have been evaluated, and a new DMA solution tailored for hyperspectral images has been proposed. Results show that the proposed DMA core, CubeDMA, handles targeted memory access patterns in more efficient manner than existing solutions while being resource efficient.

Keywords: Direct Memory Access, DMA, Hyperspectral imaging, HSI cube, on-board processing

*Corresponding author
Email address: milica.orlandic@ntnu.no (Milica Orlandić)

1. Introduction

Hyperspectral sensors for satellite application are capable of generating high-dimensional imagery with detailed information about the sensed scene. A push-broom scanner is widely used for obtaining these images by simultaneously collecting spectral information in a line-by-line fashion. Hyperspectral images (HSI) or HSI cubes are characterized by three dimensions (N_x, N_y, N_z) - spatial resolution (number of pixels), temporal-spatial resolution (frame rate) and spectral resolution (number of bands) as presented in Fig. 1. A component in the HSI cube is specified by coordinates (x, y, z). An HSI pixel with fixed (x, y) coordinates consists typically of hundreds of components in spectral domain, whereas an HSI frame is defined by a set of samples for one y coordinate. Sample ordering is a mapping of the cube components to a unique one-dimensional index from three-dimensional coordinates. The most common sample orderings are Band Interleaved by Pixel (BIP), Band Interleaved by Line (BIL) and Band Sequential (BSQ) presented in Fig. 2. In BIP ordering, each full pixel is accessed sequentially, starting at the upper left pixel and traversing the cube line by line towards the lower right pixel. In BIL ordering, traversing is performed frame by frame in (z, x) order. In BSQ ordering, the components are traversed band by band corresponding to (z, y, x) mapping order.

Satellite onboard processing systems have emerged as an attractive solution to deal with high computational requirements and to ensure fulfillment of real-time constraints in the processes of data acquisition, data interpretation and decision making. A current trend for onboard systems is the use of hybrid processing systems - System-on-a-Chip (SoC) platforms. These SoC devices adapt and combine computing architectures such as CPUs, GPUs, FPGAs and DSPs, where the advantages of each technology are used in partitioning process of the algorithms. In particular, the SoC systems with reconfigurable hardware have become the standard choice for onboard remote sensing applications and compression due to small size, weight, power consumption and resistance to damages and malfunctions caused by ionization radiation [1]. An important as-

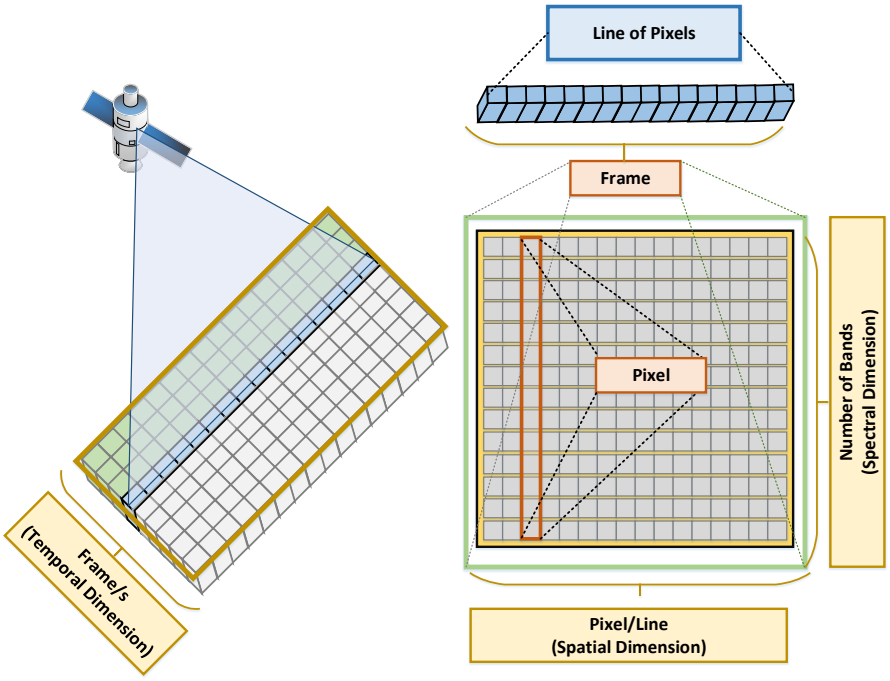


Figure 1: Hyperspectral cube acquisition process

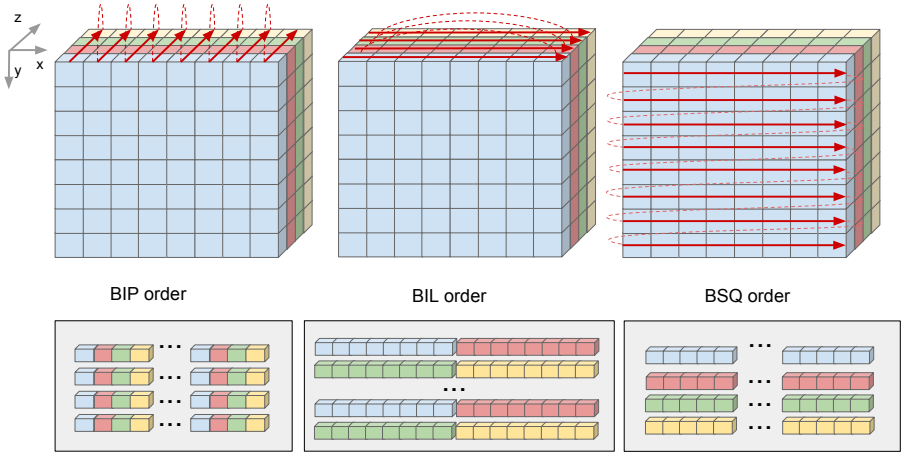


Figure 2: Sample orderings of HSI cube

pect in SoC development is to establish fast communication and high data rates between different portions of hardware, which is often the main bottleneck of a system processing large data sets. One of the most efficient ways for streaming data from the memory is by establishing direct communication between the memory and the processing core on the FPGA and by excluding the Central Processing Unit (CPU) from this critical path. This is achieved by using Direct Memory Access (DMA) cores.

In satellite applications, HSI data cubes fetched by push-broom scanners are usually arranged in the memory in BIP ordering. Due to this fixed memory data arrangement, it is required from the system to support in an efficient manner other data access patterns used by the onboard processing algorithms. The HSI-related algorithms can be divided into two distinct categories: pure HSI compression systems and HSI custom-application processing such as target detection, classification or anomaly detection. Some of these algorithms process a whole cube, while others work on smaller sub-cubes (blocks) in spatio-temporal (x, y) or spatio-spectral (x, z) planes. The existing processing cores of algorithms from both categories are often built within larger systems with standardized communication protocols. These systems are explored to detect input access patterns of interest and streaming capabilities in terms of throughput. In that context, a recent hardware implementation of PCA dimensionality reduction for hyperspectral images [2] proposes supporting communication system based on PLB bus and PLB DMA [3]. An FPGA-based hardware implementation of the CCSDS-123 compression algorithm with BSQ ordering is proposed in [4], where a supporting system based on DMA communication is also described. The need for fast block-based data streaming arises in widely used compression standards such as JPEG2000 and JPEG. The JPEG2000 standard employs the discrete wavelet transform (DWT) and defines tiles (blocks) with sizes which vary up to a complete image, whereas the JPEG standard performs DCT transform on fixed 8×8 blocks.

For other onboard HSI remote sensing applications, the custom product, such as the map of pixels indicating the spectral signature of interest in target

detection, is downlinked. An FPGA implementation of the HySime algorithm [5] for the determination of the number of endmembers in a given scene describes the communication with the memory achieved through PLB DMA on a PLB bus where the IO overhead is reduced by FIFOs for ensuring correct data reading/writing. An FPGA implementation of the CEM algorithm and RX anomaly detector [6] implements streaming background statistics (SBS) input FIFOs for streaming HSI data from the memory and uses a DMA core in combination with a PCIe bus to send data to PCI peripheral for reading. An Automatic target detection algorithm (ATPG) implementation [7] uses a DMA core in two-directional communication with memory where a structure with FIFOs is used for data prefetching of the pixel input stream.

In the context of these applications, the communication system of an onboard HSI processing system is required to support:

- Capability of streaming a HSI image in BIP or BSQ ordering,
- Capability of streaming a HSI image in block-wise BIP and BSQ ordering,
- Support for component size other than byte multiples.

Thus, in this paper, available DMA cores are studied with respect to these requirements. Existing DMA solutions are found not to be suitable for a number of scenarios. In particular, the limitations are found in the scenarios with the large HSI data set characterized by pixel bit-widths which are not byte multiples and with limited on-board memory which requires the data to be stored as compactly as possible. In order to overcome these limitations, an efficient and fast on-chip communication core *CubeDMA* suitable for HSI data access patterns of interest is proposed.

The paper is structured as follows: existing FPGA-related DMA cores, their interfaces and building blocks are presented in Section 2. The proposed high-throughput DMA implementation is described in Section 3. Results are summarized in Section 4. Finally, the conclusions are given in Section 5.

2. Direct Memory Access

Data streamed from sensors and variables required for data processing are usually stored in the memory. High-speed memory access is a critical feature for a number of systems which process large data sets due to the increase of memory-bound applications. DMA cores are commonly used to perform data exchange in a shared memory system without involving the CPU so that the CPU is allowed to perform other tasks during the data transfer.

2.1. State of the Art

A conventional Direct Memory Access (DMA) controller performs a transfer of contiguous memory locations supporting only simple data access patterns. However, this technique is not efficient for accessing non-contiguous memory locations (complex memory patterns) as it introduces substantial delays and lowers the throughput of the complete system.

Most of the recent works with respect to direct communication with memory propose complex descriptor-based DMA architectures with scratchpad memory to support both regular and irregular data patterns such as streaming, linked list, and tree-based data transfers for image/video applications. However, none of the researches concretely focuses on HSI-related streaming and requirements imposed by HSI applications.

The Xilinx AXI DMA [8] is highly configurable core by tuning a large number of parameters. The core works in two modes - a slimmed-down basic mode and a descriptor-based scatter-gather mode. This core is used often as a building block of a more complex DMA engines on Xilinx SoCs. The AXI Video Direct Memory Access (AXI Video DMA) [9] core is a specialized soft core that provides high-bandwidth direct memory access between memory and video-related peripherals. Programmable Pattern-based Memory Controller (PPMC) [10] supports data intensive applications having regular access patterns (vector and tiled access patterns) by the use of multiple descriptor blocks. Advanced Pattern based Memory Controller (APMC) [11] includes a specialized descriptor-based

DMA engine supporting both regular and irregular memory access patterns (load-store, streaming, array, linked list, and tree based data transfers) targeting applications such as 3-D stencil algorithms, linked list buffer or binary tree based Huffman coding. The data rearrangement engine (DRE) [12] performs in memory data restructuring to accelerate irregular, data-intensive applications. The DRE core consists of a programmable DMA unit (a load/store unit) along with a microcontroller which executes a simple set of application directed commands. Finally, CoRAM++ [13] is a programming environment for FPGA computing supporting complex data structures such as multidimensional arrays and linked lists. It is based on a set of memory interfaces for each supported data structure and provides a specialized soft-logic implementation of the datapath to memory. In the context of HSI processing, particular interest is found in multidimensional array structures, where array traversal order and data layout can be selected at run time. Supported data layouts in memory are common row-major order (BIP ordering) or tiled order (block-wise ordering). For data stored in row-major order, strided data accesses during column (BSQ ordering) traversal has been shown inefficient.

2.2. AXI DMA cores and HSI data streaming

In this section, the available Xilinx DMA cores, AXI DMA and AXI Video DMA, are described and analyzed in the context of 3-D HSI data access patterns.

Recent Xilinx SoCs use a set of open bus standards, Advanced eXstensible Interface (AXI) [14] for connecting master and slave components within large systems. The protocol is based on a common handshake mechanism for data transfer controlled by two signals. The first signal indicates presence of valid data on the data lines and the second signal is asserted by the recipient when it is ready to receive the data. The transfer is acknowledged when both signals are asserted in the same clock cycle, a *beat*. The beat is the fundamental unit of data transfers and one transaction can consist of several data beats. Instead of initiating a new transaction for every unit of data, a burst transfer is introduced defining a *burst size* and *burst length* in addition to the address. The *burst size*

indicates the number of bytes to be transferred in one beat, whereas the burst length provides a number of beats per transaction.

The AXI DMA core is configured directly by dedicated registers accessible through the CPU's memory map. The CPU sets a start address and a length of contiguous memory locations and controls transfer completion either by polling or by the use of interrupts. However, many applications require to gather data from separate locations. The AXI DMA core performs this operation in *scatter-gather* mode defining the DMA configuration which relies on a chain of block descriptors (BDs) for the transfer description. Each BD contains a start address, a length and several flags (e.g. whether the block is the last one). Fig. 3 shows a transfer of several scattered blocks by the use of block descriptors. The use of a DMA without scatter-gather mode in such scenarios requires the CPU to set up the transfer of each contiguous block introducing delays and limiting the efficiency of system partitioning.

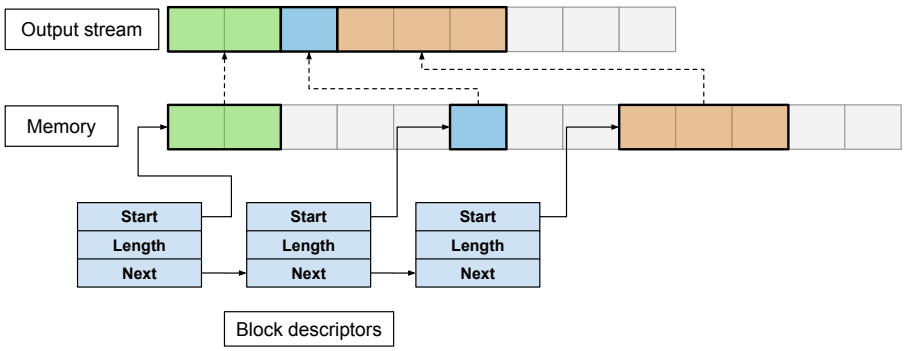
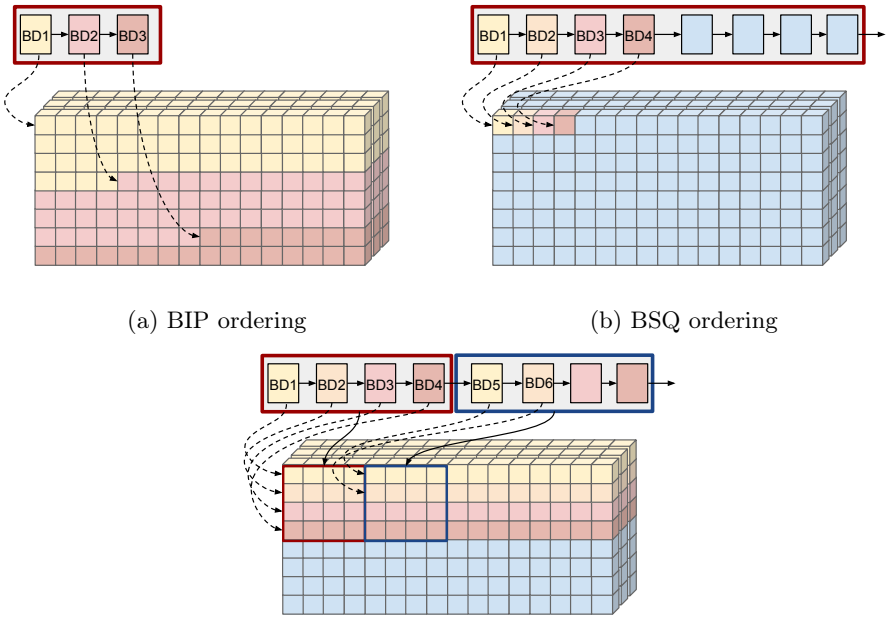


Figure 3: Scatter-gather transfer using a chain of block descriptors

The AXI DMA performs a transfer of a complete HSI cube in sequential BIP ordering by creating a BD chain as shown in Fig. 4a. In the BD chain, all but the last block descriptors have a maximized length field which depends on configuration/generics of the AXI DMA, and the last block descriptor contains a length field set to the remaining number of bytes. A transfer in BSQ ordering is required to be performed by using only one block descriptor per pixel component

(Fig. 4b). In this manner, a BD chain with length equal to the number of components of the whole HSI cube introduces unacceptable overhead and delay related to operation of fetching block descriptors. Some applications require data to be fetched from scattered locations with a constant access pattern such as rectangular sub-blocks. In this context, block-wise BIP ordering requires one BD per block row as presented in Fig. 4c.



(a) BIP ordering

(b) BSQ ordering

(c) Block-wise BIP ordering

Figure 4: Creation of BD chain for different sample orderings

The configuration of AXI DMA for operating in 2D mode has the BD length field replaced with fields of a vertical size ($VSIZE$), a horizontal size ($HSIZE$) and a stride. The horizontal size equals the number of bands to transfer in parallel, whereas stride field contains information about the number of bands to skip/jump. The vertical size is the number of contiguous memory transfers to perform. For example, to transfer first four bands in BSQ ordering, the start address of the transfer is set to the start address of the cube and field values are set as $HSIZE = 4$, $Stride = N_z$ and $VSIZE = N_x \times N_y$. AXI DMA in

2D mode, however, does not support unaligned transfers i.e. the transfers are required to start at an address that is divisible by the native word length used for memory access.

The AXI Video DMA [9] is designed for video streaming, where a video consists of a sequence of still-images. The video frames, in this case, are defined as contiguous areas in memory which contain the pixels of a particular still-image. The AXI Video DMA contains registers available to configure transfers of up to 32 frame buffers. The frame buffers are pointed to with an address. Horizontal size, vertical size and stride parameters are supplied for each frame buffer, where the horizontal size and stride parameter fields have 16-bit limitation. This limitation can affect storing of HSI images by breaking HSI frames into different frame buffers. In BIP ordering, Video DMA frames are defined as areas of the size $N_x \times N_z$. For BSQ transfers, a Video DMA frame contains only one component and the $VSIZE$ field must be then set to $N_x \times N_y$ which can easily exceed field range for large cubes. This implies several separate transfers and CPU interventions for the transfer of complete cube.

A short summary of the transfer modes for existing DMA cores and their limitations for data patterns of interest is presented in Table 1.

The described Xilinx DMA cores use the AXI DataMover [15] to perform the actual data movement from/to memory. The DataMover is not a standalone block and it requires a register interface for CPU control and external logic to combine several DataMover transfers to cover the whole HSI cube. A block diagram of DataMover is presented in Fig. 5. The DataMover core accepts simple commands to indicate data source and amount of data for transfer and to perform the transfer. After processing a command, a status word is sent as a response. Command and status words are buffered in a FIFO to ensure minimal delay between their execution.

Table 1: Existing Direct Memory Access Modes

AXI DMA - Scatter-Gather Mode	
BIP	a BD chain of a length necessary to describe the whole transfer
Block-wise BIP	a BD chain with one descriptor per row in the block.
BSQ	a BD chain with one BD per component - Not usable for HSI processing
AXI DMA - 2D Mode	
BIP	a chain of two BDs, the first BD with HSIZE = STRIDE = maximized and VSIZE=$(N_x \times N_y \times N_z)$/HSIZE, and the second BD with leftovers from the first transfer (Image start at 64-bit aligned address) - Not usable for HSI processing
Block-wise BIP	a chain of BD per row, with HSIZE = STRIDE = $(block\_width \cdot N_z)$ and VSIZE = 1. Block start at 64-bit aligned addresses - Not usable for HSI processing
BSQ	Each BD with a STRIDE=N_z, HSIZE which indicates the number of bands to extract and VSIZE=$N_x \times N_y$ - Not usable for HSI processing
AXI DMA - Video Mode	
BIP	16-bit limitation of HSIZE and STRIDE fields, 32-frame buffer - Not usable for HSI processing
Block-wise BIP	16-bit limitation of HSIZE and STRIDE fields, 32-frame buffer - Not usable for HSI processing
BSQ	One-component video frames, limited to 32-frame buffer - Not usable for HSI processing

3. CubeDMA Implementation

Among the available DMA solutions for SoCs with FPGAs, none of the cores fulfills given transfer requirements for 3D HSI data at the sufficient level. The AXI DMA can perform data transfers in BIP and block-wise BIP orderings, but challenges arise for BSQ transfers. The efficiency of AXI DMA in 2D mode is limited by the requirement of 64-bit address alignment, whereas the Video DMA

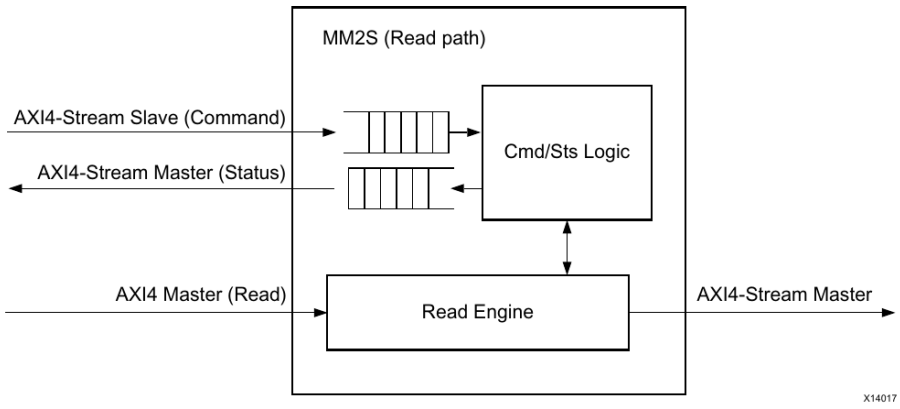


Figure 5: Overview of the DataMover core [15]

is not flexible enough to be used for HSI images. For these reasons, a custom DMA core, *CubeDMA*, specialized for hyperspectral image streaming patterns with various bit-widths is implemented. The dimensions (*width*, *height*, *depth*) corresponding to (N_x, N_y, N_z) of the HSI cube, as well as the block dimensions (*block_ width*, *block_ height*) for the chosen HSI data processing algorithm are set before the processing starts. In the implementation, there are no restrictions for the HSI cube size - the cube width N_x and height N_y do not need to be divisible by the *block_ width* or *block_ height*. This implies that the last block in each block row can have a width less than the *block_ width*, and the last row of the blocks can contain blocks of a height that is lower than *block_ height*. However, the block dimensions are restricted to power of 2 as it is well-aligned with the requirements of the most block- and tile-based algorithms. The introduction of block dimension constraint allows a number of computations to be simplified, so multiplication and division by the block dimensions become shifting, whereas residual computation (modulo operation) becomes least significant bits assignment. Regardless of BSQ or BIP ordering, the CubeDMA can order the pixels in sequential or block-wise manner. The sequential transfer starts at the first pixel (upper left) and proceed through the cube line by line until the last pixel (lower right). In a block-wise transfer, the cube is divided into blocks with

block_height and *block_width*. The transfer orderings of pixels within a block and blocks within a cube are shown in Fig. 6. The red arrows indicate the pro-

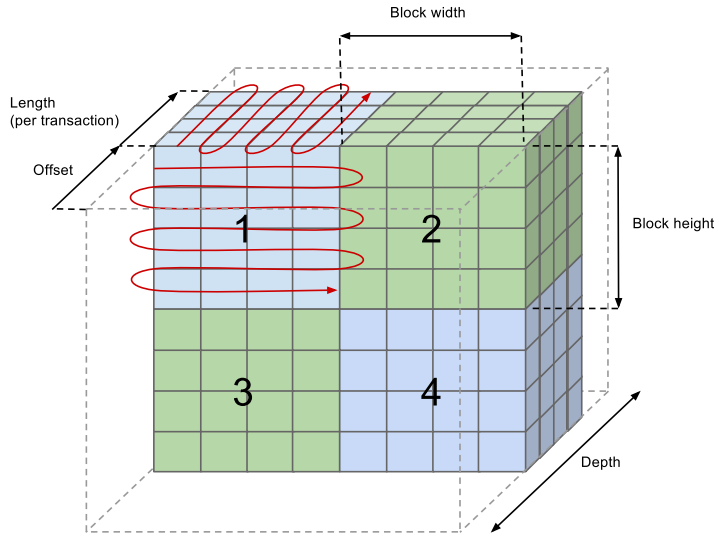


Figure 6: Processing order for a block-wise transfer in the CubeDMA

cessing order of the pixel components within the blocks, whereas the numbers indicate the processing order of blocks within the cube. The band coordinate z which block transfer starts at is assigned to parameter *offset*, whereas the *length* per transaction is the number of contiguous bands to transfer. The CubeDMA core consists of two streaming data channels, Memory Map to Stream (MM2S) and Stream to Memory Map (S2MM), and the focus of the paper is to describe the MM2S channel for data streaming from the memory to the processing core. An overview of an MM2S channel with its building units Register interface, DataMover, Controller and Component unpacker is given Fig. 7.

3.1. DataMover

The DataMover IP [15] performs the transfer from/to the memory by AXI transactions starting at a given address and with a given length as the basic unit of data transfer. Depending on the cube dimension and component bit-width, the DataMover transfers a set of components (parts of a pixel), whole

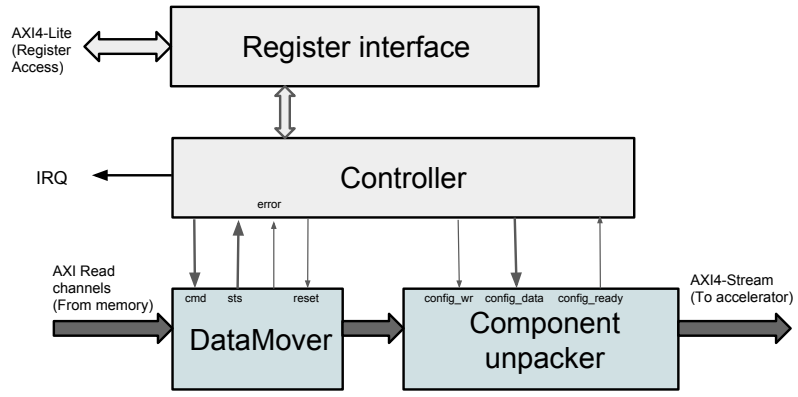


Figure 7: Architecture of the CubeDMA MM2S channel

pixels or rows of pixels. Transfer progress is controlled by the *Completion tally* module, a counter which keeps a tally of the number of issued commands and received status words. The counter is incremented/decremented each time a command/status is handshaked.

3.2. Register Interface

The Register interface exposes control and status registers to the CPU's memory map in order to configure the core. The register layout is detailed in Table 2 and a number of the register parameters are illustrated in Fig. 8. The parameters such as the size of one row in number of components, number of band transfers per pixel and number of components per row in the last block are used for address computation.

3.3. Controller

The Controller issues commands to the DataMover for the traversal of the HSI cube in the manner defined by the parameters in the registers. Each time the DataMover is ready to accept a new transaction, the Controller computes the byte address of the first component and the number of bytes to transfer

Table 2: Register layout for the CubeDMA

Field [Unit]	Description	Bits
Control and length register (0x00)		
Start	Transfer starts on bit transitions from 0 to 1	0
Block-wise mode	Cube is read in blocks of specified size	2
BSQ mode	Cube is read in BSQ mode + Number of bands in parallel	3
Error IRQ Enable	IRQ is triggered when error condition arises	4
Completion IRQ Enable	IRQ is triggered when transfer is completed	5
Length [comp]	Number of bands to transfer	15-8
Start offset [comp]	Band number transfer starts from	23-16
Status register (0x04)		
Transfer done	Indicates whether the transfer is completed	0
Error mask	Indicates which errors occurred	3 - 1
Error IRQ flag	Indicates when IRQ is triggered due to error	4
Completion IRQ flag	Indicates when IRQ is triggered due to completion	5
Base address register (0x08)		
Base address	The address of the first component in the first pixel	31 - 0
Dimension register 1 (0x0C)		
Width [pixels]	The width of the HSI cube	11 - 0
Height [pixels]	The height of the HSI cube	23- 12
Depth [comp]	Lower 8 bits of the depth of the HSI cube	31- 24
Dimension register 2 (0x10)		
Block width	$log2$ of the width of each block	3-0
Block height	$log2$ of the height of each block	7-4
Depth [comp]	Upper 4 bits of the depth of the HSI cube	11-8
Last block row size [comp]	Number of components within each row in the last block	31 - 12
Row size register (0x14)		
Row size [comp]	Number of components in one row of the cube	19-0

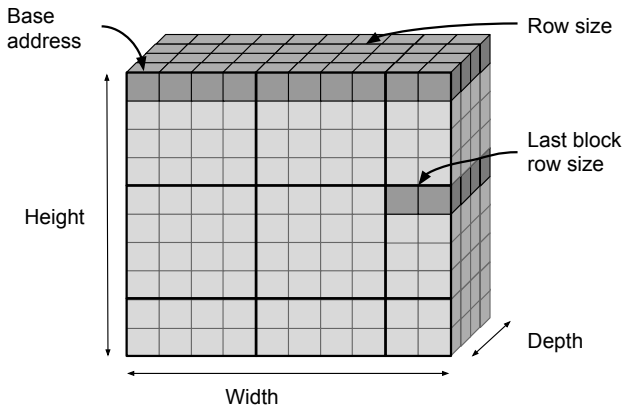


Figure 8: Register parameters

in that transaction. The architecture of the Controller module is illustrated in Fig. 9. The Controller performs the following operations:

- control of the operation sequencing - (State machine unit),
- generation of the component addresses and conversion of the HSI cube components to byte units - (Address generator unit),
- control of the command/status operations.

Operation sequencing is controlled by the state machine consisting of six states as presented in Fig. 10. In the *running* state commands are issued to the DataMover, whereas in the *wait_ complete* state the controller awaits for the last command to be processed. In the *hard_ error* state, the DataMover signals an internal error which requires reset by moving to the *reset* state. The DataMover reports a status word with an error bit set in *sts_ error* state.

The address generation logic consists of a set of counters used for computation of component addresses. The logic driving these counters is computed based on parameters:

- `num_blocks_x`, `num_blocks_y` - number of blocks in total in x and y dimension respectively,

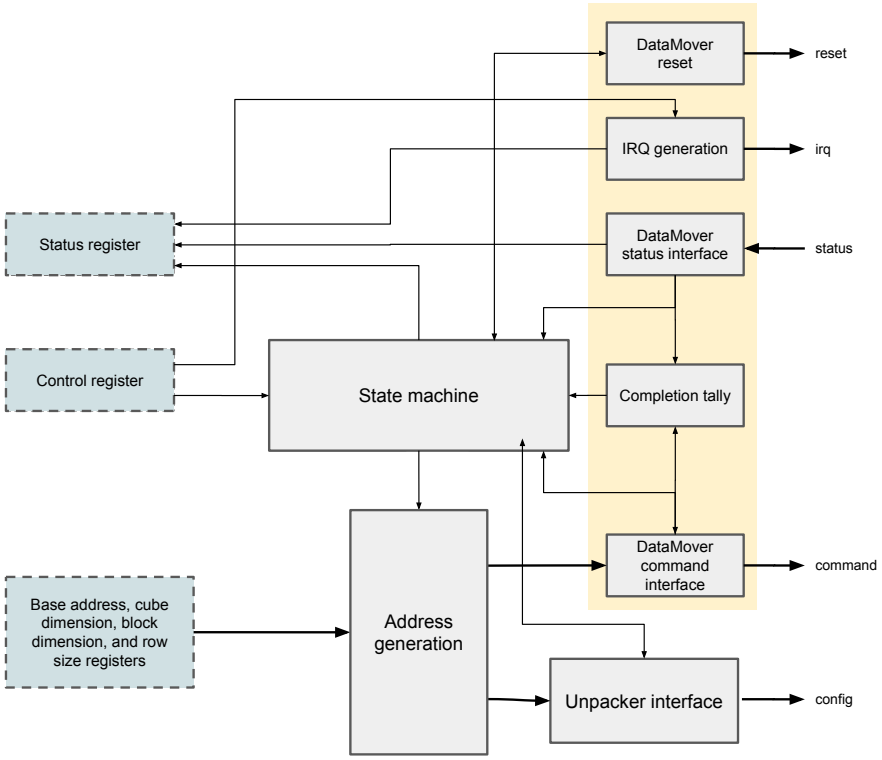


Figure 9: Architecture of the CubeDMA Controller module

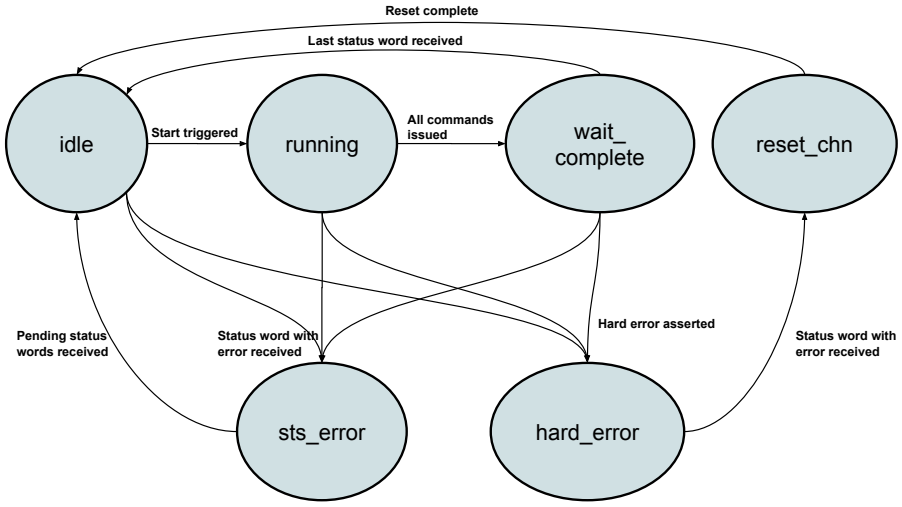


Figure 10: The state transition diagram for the State machine unit in the Controller

- `curr_blocks_x` and `curr_blocks_y` - the currently processed blocks in x and y dimension respectively,
- `h`, `w`, `block_h`, `block_w`, `rsize` and `offset` related to the fields *height*, *width*, *block_height*, *block_width*, *row_size* and $offset$ of the registers in the Register interface.

The pseudo-code for address generation is presented in Listing 1. The position of the current block within the cube (`curr_blocks_x` and `curr_blocks_y`) and the `offset` are updated for each band and block. The block address (`block_addr`), the address of the complete row (`row_addr`), the start address of each row of the block (`block_r_addr`) and the component addresses (`comp_addr`) are computed for each block.

```
for num_plane_transfers-1 to 0:
  block_addr, row_addr,  block_r_addr, comp_addr = offset

  for block_y in num_blocks_y-1 to 0:
    for block_x in num_blocks_x-1 to 0:
      for y in curr_block_h-1 to 0:
        for x in curr_block_w-1 to 0:
          if mode_block:
            if block_x = 0:
              curr_block_w = w mod 2**block_w
              length = last_block_row_size
            else:
              curr_block_w = 2**block_w
              length = 2**block_w * depth
            if block_y = 0: curr_block_h = h mod 2**block_h
            else:    curr_block_h = 2**block_h
          issue command(comp_addr, length)
          if x = 0 and y = 0:
            if block_x != 0:  block_addr += w mod 2**block_w
            else:
```

```
                block_r_addr += 2**block_h*rsize
                block_addr += block_row_addr
            if x != 0: comp_addr + = depth
            else:
              if y != 0:  row_addr += w
              else: row_addr = block_addr
              comp_addr = row_addr
            wait for tick from state machine
    offset = offset + comp_per_cycle
```

Listing 1: Pseudo-code of address generation process

The controller operates internally with components as the fundamental units. Since the component size is not necessarily a multiple of a byte, it is required to perform translation from component addresses into byte addresses. The translation of a component address into a *byte address* and an *offset* for 10- and 12-bit components is illustrated in Fig. 11. The *byte address* is computed by multiplying the component address and the number of bits per component BPC followed by byte-division as follows:

$$byte\ address = \frac{BPC \cdot component\ address}{8}, \tag{1}$$

whereas the *offset* is given as:

$$offset = (BPC \cdot component\ address) \bmod 8. \tag{2}$$

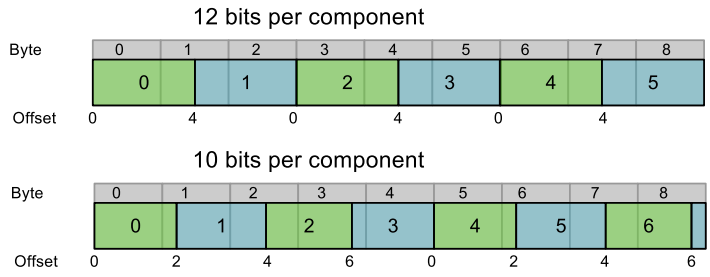


Figure 11: Component address, byte address and offset

Conversion from number of components to a number of bytes N_{bytes} is also required. For a given BPC, the number of bytes to represent n components is given by:

$$N_{\text{bytes}} = \frac{n \cdot BPC}{8}. \tag{3}$$

The number of bytes N_{bytes} depends also on the *offset* of the starting component. Fig. 12 shows that in the first two scenarios four bytes are required for the transfer of three 10-bit components, whereas in the third example the most significant bits of the last component are shifted across a byte boundary by the *offset* requiring an extra byte to be transferred. For this reason, the offset is included in the computation of N_{bytes} as follows:

$$N_{\text{bytes}} = \left\lceil \frac{\text{offset} + n \cdot BPC}{8} \right\rceil. \tag{4}$$

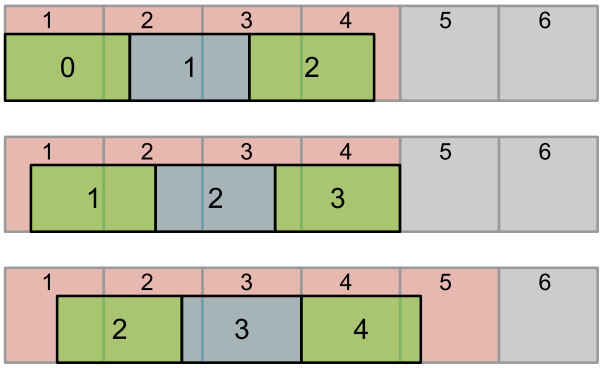


Figure 12: Number of bytes required to transfer three 10-bit components starting at various offsets

3.4. Component Unpacker

When the HSI cube is packed in the memory with component sizes that are not byte-multiples, components can be split across two (or more) bytes with their LSB bit at an *offset* within a byte. The component unpacker alters the data stream from memory so that user-defined number of complete components

N_{comp} are input to the processing core. An example of component unpacking process is presented in Fig. 13 with selected parameters - $BPC = 12$ bits, $N_{\text{comp}} = 4$ and length of input word $N = 64$ bits.

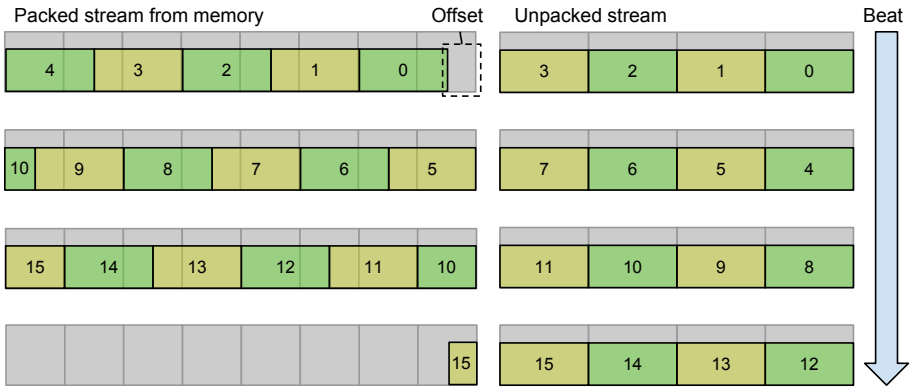


Figure 13: An example of packed data stream coming from memory and the resulting unpacked data stream

The structure of Component unpacker module is presented in Fig. 14 where each stage requires configuration data associated with the incoming packet. The configuration data consist of a number of parameters useful for the unpacking process such as the number of bits to shift the incoming word from memory in the offset shifter and the number of valid components in the last packet from memory. The start of a new packet is detected in packet detector stage simultaneously with configuration data sent into a FIFO from the controller module.

The offset shifter performs a shifting operation of the N-bit incoming data packet to remove the offset computed by Eq. 2. If the first component in a transfer starts at a non-zero offset n_{offset} in memory, the incoming data is shifted n_{offset} positions and the most significant bits are filled with the n_{offset} least significant bits from the next data packet. The shifting, which propagates through the subsequent data packets in the transfer as presented in Fig. 15, is implemented in hardware by storing each incoming data word in a register. The $(N - n_{\text{offset}})$ most significant bits in the register are used as the least significant

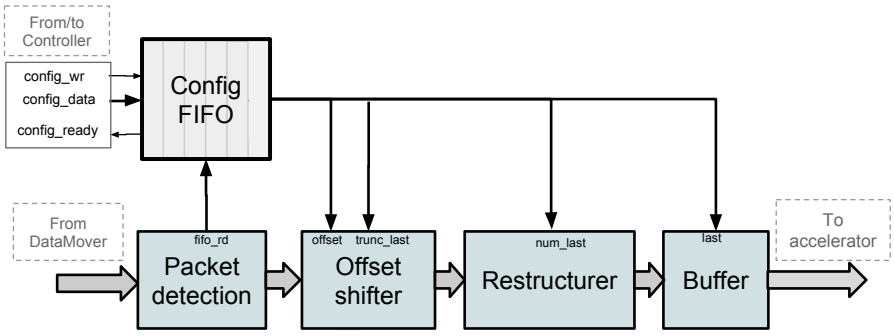


Figure 14: The Component unpacker module

bits of the output, and the least significant n_{offset} bits of the current data are used as the most significant n_{offset} bits of the output.

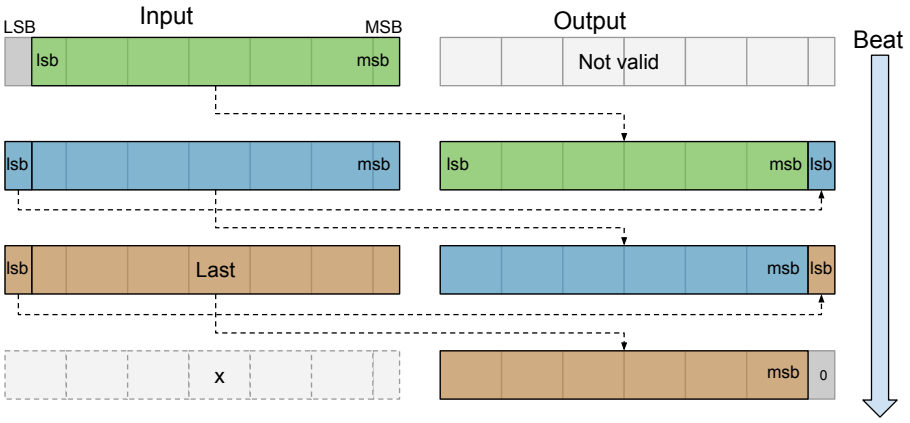


Figure 15: Behavior of the offset shifter

The component restructurer reorganizes N-bit input data packet into a maximum number of components N_{max} and a set of *leftover* bits of a component, where the maximum number of components is computed as:

$$N_{\text{max}} = \left\lceil \frac{N}{BPC} \right\rceil . \tag{5}$$

The complete components are forwarded, whereas the leftover bits are stored in a register to be combined with the components in the next data word. To maintain

the throughput from the DataMover, the Component restructurer outputs in average a number of bits equal to the the incoming number of bits. Since the constraint is to output complete components, the restructurer outputs data widths of $N_{\max}+1$ components. For instance, for 10-bit components and 64-bit input data stream, the number of components contained in the input stream is $N_{\max}=6$. This introduces a requirement for the output data stream to have a width of 7 components. The example is illustrated in Fig. 16. The number of

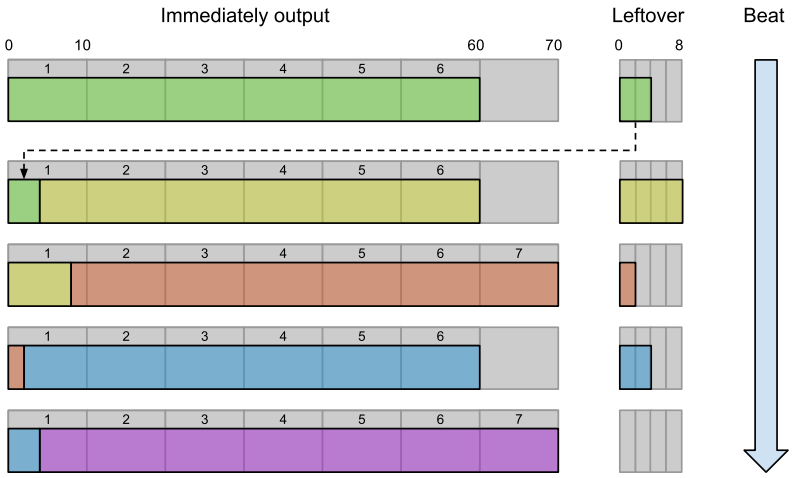


Figure 16: Example of a five-cycle restructuring process for 10-bit components and 64-bit input stream

components to output and the number of leftover bits to put into the register are different in each step. However, there is a repeating pattern and in the example of 10-bit components and 64-bit input stream, the pattern contains 5 different steps. The number of steps in a pattern is the lowest number of N-bit words required to get a multiple of the component width:

$$steps = \frac{\mathrm{lcm}(N, BPC)}{N}, \tag{6}$$

where `lcm` is least common multiple operator. The implementation uses a counter to keep track of the current step. Based on the step count, a set of multiplexers determines arrangement of the output data from the leftovers and

input data, and decides the next value for the leftover register.

Finally, the buffer collects the components from the restructurer and outputs N_{comp} components in parallel to the processing core. The buffer is implemented as a FIFO of the depth of $N_{\text{buff}} = N_{\text{max}} + N_{\text{comp}} - 1$, where the number of components N_{comp} varies in the range $[0, N_{\text{max}}]$. For instance, for values of user-defined parameters $N = 64$, $BPC = 10$ and $N_{\text{comp}} = 4$, the depth of the buffer N_{buffer} is 10 components. The buffer accepts data as long as it is not full or in the case a read operation is accepted and the free buffer space after that read operation is sufficient.

In the block-wise ordering, additional challenge arises when the output components from the Component unpacker originate from different blocks in the HSI cube. In that case, it is required to identify relations between components and blocks. In order to track the components from different blocks and to ensure synchronous transfer of control and data signals, a set of two control signals per component are concatenated to the components in the component buffer. These control signals indicate whether the component is part of the last pixel in a block in the last column of blocks and whether the component is part of a block in the last row of blocks.

4. Results

The majority of HSI algorithms in satellite applications process HSI cubes in a variety of data access patterns such as BIP, BSQ, block-wise BIP and block-wise BSQ ordering, whereas the cube data streamed from a HSI sensor in a push-broom imager are usually packed in a sequential manner (BIP) in memory. To achieve high throughput when streaming HSI cube data from/to memory, a specialized DMA controller, CubeDMA, is implemented. In this manner, a fast and simple communication layer between HSI processing cores on FPGA logic and memory is established. The custom design of the CubeDMA core allows, unlike in the AXI 2D DMA or AXI Video DMA, the register fields to be large enough to perform the transfer without errors or convoluted workarounds.

The proposed architecture of the CubeDMA core is described by the VHDL language. The Xilinx Vivado tool is used for synthesis, implementation, power estimation and verification of the proposed architecture on a Zedboard development board which combines ARM processor cores with a Zynq-7020 FPGA. The estimated maximal operating frequency of CubeDMA core is approximately 132 MHz and the critical path is in the address computation logic. The system presented in Fig. 17 containing CPU, DDR memory, CubeDMA and processing cores is used in the verification phase. The operating frequency f_{oper} of the proposed system is constrained to 100 MHz in the verification process and the following performance analysis. The data streaming through CubeDMA

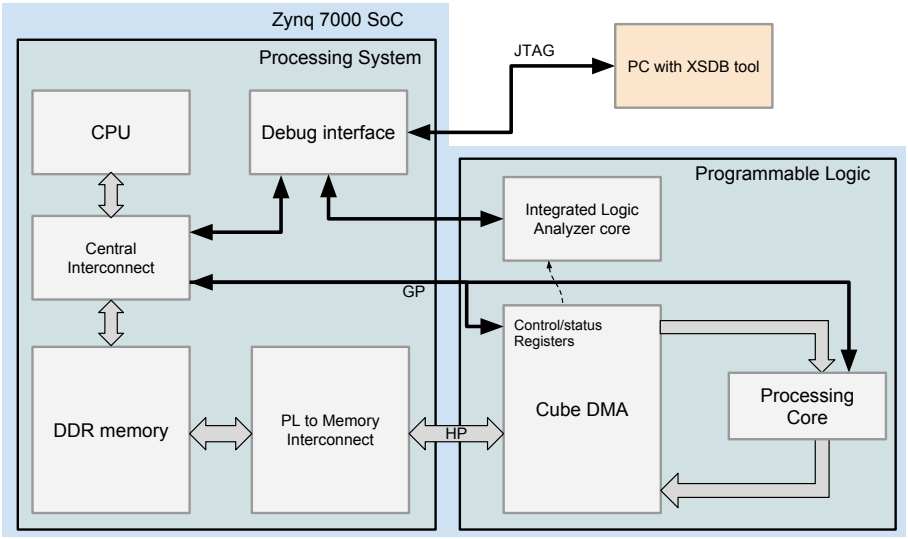


Figure 17: Overview of Zynq-7000 system for testing CubeDMA

is functionally tested in the presented system by establishing communication between memory and a custom processing core. The choice of processing core has no impact on performance of the CubeDMA and this is initially verified by setting as the processing core a FIFO module which is always ready to accept new data each clock cycle. In the later stage, an HSI-related CCSDS-123 compression core [16] is set instead of the FIFO module. The system has been tested by streaming a real HSI image of a size $512 \times 2000 \times 128$ col-

lected by HICO imager [17] for a variety of generic parameter values such as $[N_{\text{comp}}, BPC] = \{[1,16],[4,16],[5,12],[6,10]\}$ and $N = 64$ bits. The resulting bitstreams for each $[N_{\text{comp}}, BPC]$ pair streamed by the CubeDMA from CCSDS-123 core back to the memory have been successfully compared to the compressed bitstreams generated by the CCSDS-123 reference software Emporda [18].

The proposed CubeDMA is a core with the following generic parameters - input data width N, component bit-width BPC and number of parallel output components N_{comp}. Re-synthesis of the design is required if any of these parameters is modified. On the other side, HSI image resizing and choice of transfer mode do not affect the design since the image dimension and transfer mode parameters are modified through the register interface. Fig. 18 and Fig. 19 show how the configuration of the generic parameters impacts area utilization of CubeDMA in terms of the LUTs and registers, respectively. It is observed that the number of LUTs depends on the number of steps given by Eq. 6 in Component restructuring unit. For selected number of components per beat (8, 10, 12, 16 and 18), there are (1, 5, 3, 1, 9) number of steps respectively implying that design with 10 components uses more LUTs than with 12 components. The register utilization scales linearly with BPC. The estimated power is 0.143 W and 0.15 W for parameters $[N_{\text{comp}}, BPC]$ set to $[4,16]$ and $[5,12]$ respectively and the increase is due to the unpacking operation of non byte-multiple components.

In more details, the area utilization is examined for four modules - the DataMover, Register interface, Controller and Unpacker. The area utilization for DataMover and Register interface remains fixed regardless of configuration of the core. The resources for these two modules are presented in Table 3.

Table 4 shows the logic utilization of the Controller and Unpacker units instantiated with different component widths BPC and number of output components N_{comp}. The results show that the resources in terms of LUTs and registers for the Controller vary slightly. There is an increase when component width is not a byte-multiple due to the extra logic for the offset computation

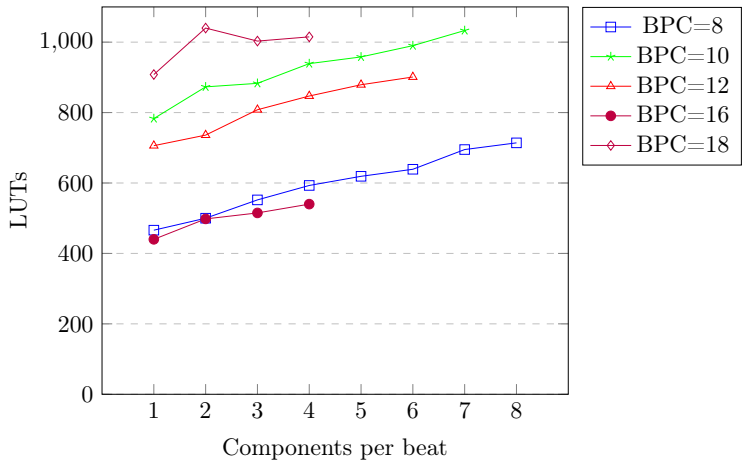


Figure 18: LUT usage for different BPC as a function of N_{comp}

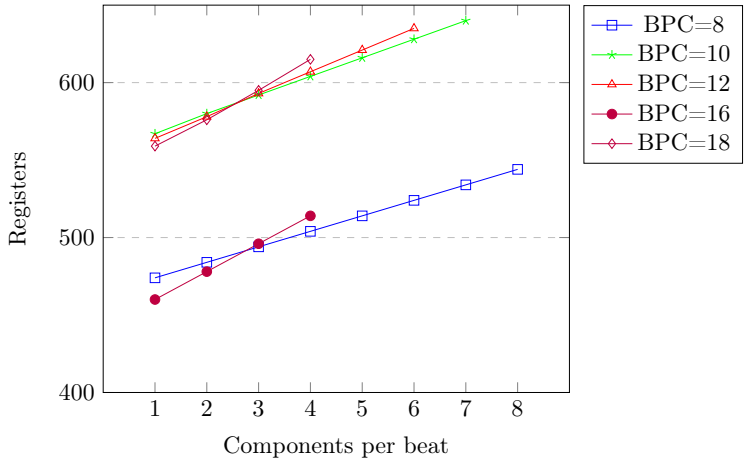


Figure 19: Register usage for different BPC as a function of N_{comp}

Table 3: Area utilization of modules and IPs whose area is independent of generic parameters

Module	LUTs	Registers
DataMover IP	922	783
Register Interface	696	465
Unpacker FIFO	78	74

Table 4: Area utilization in terms of component widths and number of components per beat

BPC	N_{comp}	Controller		Unpacker		Total	
		LUTs	Regs	LUTs	Regs	LUTs	Regs
8	1	333	248	133	226	466	474
8	2	331	248	169	236	500	484
8	3	333	248	219	246	552	494
8	4	351	248	242	256	593	504
8	5	334	248	285	266	619	514
8	6	331	248	308	276	639	524
8	7	334	248	361	286	695	534
8	8	326	248	388	296	714	544
10	1	435	250	348	317	783	567
10	2	431	250	442	330	873	580
10	3	435	250	448	342	883	592
10	4	454	250	485	354	939	604
10	5	436	250	522	366	958	616
10	6	436	250	554	378	990	628
10	7	436	250	597	390	1033	640
12	1	432	249	274	315	706	564
12	2	416	249	320	329	736	578
12	3	432	249	376	344	808	593
12	4	430	249	417	358	847	607
12	5	431	249	448	372	879	621
12	6	416	249	485	386	901	635
16	1	321	243	119	217	440	460
16	2	346	243	152	235	498	478
16	3	320	243	195	253	515	496
16	4	322	243	218	271	540	514
18	1	386	233	563	326	908	559
18	2	383	233	657	343	1040	576
18	3	386	233	617	362	1003	595
18	4	388	233	627	382	1015	615

and the byte length conversion. However, the largest variations are in the Unpacker module. The Component restructurer and the component buffer are the two largest contributors, and their area utilization varies considerably depending on the configuration. The size of the restructurer is affected by the number of steps determined by the least common multiple between input word width N and the component bit-width BPC. The size of the buffer both in terms of registers and LUTs is primarily determined by N_{buffer} which scales linearly with the number of output components N_{comp}. The Component restructurer is only included when component width is not byte-multiple, whereas the buffer is always included.

With respect to on-chip memory, the CubeDMA core uses 3.5 of available 140 block RAM tiles on Zynq-7020 FPGA regardless of generic parameter. The design blocks using on-chip memory are the DataMover and the configuration FIFO. For the channel configuration, 1.5 block RAM tiles are used per DataMover instance, whereas the configuration FIFO with its depth set to 128 configuration words of 12 bits requires 0.5 block RAM tiles.

The DSP utilization depends on the parameters BPC and N. For larger BPC values, such as $BPC = 18$, the synthesis tool infers a DSP to compute the product $BPC \cdot component\_address$ in Eq. 1, whereas for configurations with $BPC = [8, 10, 12, 16]$ the DSP blocks are not used.

The analysis of the existing DMA cores, AXI DMA, AXI DMA 2D and AXI Video DMA, shows that the AXI DMA is the most flexible core for HSI streaming orderings and further performance testing of AXI DMA and CubeDMA is performed. The cube dimensions used for testing are $500 \times 2000 \times 100$ of 8-bit components stored in BIP ordering in the memory, where the component bit-width is chosen based on AXI DMA byte-multiple limitation. The parameters in AXI DMA and CubeDMA used for this comparison are set as shown in Table 5, whereas the performance results of the AXI DMA and CubeDMA for HSI data cube streaming are shown in Table 6. The theoretical throughput is computed as the product $N \cdot f_{oper}$, and for input word length $N = 64$ from Table 5, both AXI DMA and CubeDMA can achieve the theoretical throughput of 800 MB/s

in the case of BIP ordering. A speed-up of 2.3 is reported for the CubeDMA compared to the AXI DMA for block-wise BIP ordering with block size of 8×8. The overhead in AXI DMA transfers for block-wise ordering is due to the fact that the AXI DMA fetches a new block descriptor for each row in the block from memory (Fig. 4c). The CubeDMA has no overhead, but there are still reported delays related to the DDR memory controller when starting a transfer of a new row in the block.

For a transfer of the band z in BSQ ordering and given parameter values, the maximum theoretical throughput is 100 MB/s, computed as $BPC \cdot f_{oper}$, since only BPC bits of the N-bit input word, corresponding to a component in band z, are sent further to the processing core. The throughput achieved by CubeDMA in BSQ ordering is 14.1 MB/s limited by the DataMover's idle state of seven cycles between requests. The AXI DMA uses one block descriptor per component in the BSQ ordering, requiring a BD chain with the length equal to the number of components in the cube. This causes large overhead and delay related to fetching operation of block descriptors and the testing of AXI DMA for BSQ ordering has not been performed.

Results show that CubeDMA outperforms existing AXI DMA cores for streaming HSI data cubes. However, further improvements in CubeDMA performance can be introduced, in particular for BSQ ordering, by replacing DataMover module with a more efficient module.

5. Conclusion

The proposed CubeDMA core is a promising approach for streaming HSI data sets in a number of access patterns when the memory representation of the HSI cube is fixed. The optimization introduced in CubeDMA involves elimination of block descriptors for transfer description of HSI cubes and thus removal of delay overheads and simplification of the transfer setup. The flexibility of the proposed CubeDMA core is introduced through generic parameters of input data width, component bit-width and number of parallel output components.

Table 5: AXI DMA and CubeDMA parameters used in performance comparison

Cube parameters	
Width	500
Height	2000
Bands	100
Component bit-width (BPC)	8
Stored order	BIP
Block size	8×8
AXI DMA	
Scatter-Gather	Yes
Burst size	16
Stream width	64
Dynamic Realignment Engine	Yes
Length reg. size	Maximum (23 bit)
CubeDMA	
Input word width (N)	64
Components per beat (N_{comp})	8
Component bit-width (BPC)	8
Burst size	16

Table 6: Performance comparison of AXI DMA and CubeDMA for different 3-D HSI cube transfer types

	BIP		Block-wise BIP		BSQ	
	Time [s]	Throughput [MB/s]	Time [s]	Throughput [MB/s]	Time [s]	Throughput [MB/s]
Theoretical	0.125	800	0.125	800	1	100
AXI DMA	0.125	800	0.294	340	-	-
CubeDMA	0.125	800	0.129	775	7.10	14.1

The core has been extensively tested on HSI images in simulations and on a real hardware. Creating testbench infrastructure to do automated verification, inclusion of new data access patterns and further optimization towards throughput improvements are set as future work.

Acknowledgement

This work was supported by the Research Council of Norway (RCN) through MASSIVE project, grant number 270959, as well as by the Norwegian Space Center.

References

[1] S. Lopez, T. Vladimirova, C. Gonzalez, J. Resano, D. Mozos, A. Plaza, The promise of reconfigurable computing for hyperspectral imaging onboard systems: A review and trends, Proceedings of the IEEE 101 (3) (2013) 698–722.

[2] D. Fernandez, C. Gonzalez, D. Mozos, S. Lopez, FPGA implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images, Journal of Real-Time Image Processing (2016) 1–12.

[3] IBM Inc., 128-bit processor local bus architecture specifications, Tech. rep. (2007).

[4] L. Santos, L. Berrojo, J. Moreno, J. F. López, R. Sarmiento, Multispectral and hyperspectral lossless compressor for space applications (HyLoC): A low-complexity FPGA implementation of the CCSDS 123 standard, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9 (2) (2016) 757–770.

[5] C. Gonzalez, S. Lopez, D. Mozos, R. Sarmiento, FPGA implementation of the HySime algorithm for the determination of the number of endmembers in hyperspectral data, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 8 (6) (2015) 2870–2883.

[6] B. Yang, M. Yang, A. Plaza, L. Gao, B. Zhang, Dual-mode FPGA implementation of target and anomaly detection algorithms for real-time hyperspectral imaging, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 8 (6) (2015) 2950–2961.

[7] C. González, S. Bernabé, D. Mozos, A. Plaza, FPGA implementation of an algorithm for automatically detecting targets in remotely sensed hyperspectral images, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9 (9) (2016) 4334–4343.

[8] Xilinx, LogiCORE IP Product Guide, AXI DMA v7.1, Tech. rep. (2017).

[9] Xilinx, LogiCORE IP Product Guide, AXI Video Direct Memory Access v6.2, Tech. rep. (2016).

[10] T. Hussain, M. Shafiq, M. Pericàs, N. Navarro, E. Ayguadé, Ppmc: A programmable pattern based memory controller, in: ARC, 2012.

[11] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. Ayguadé, M. Valero, Advanced pattern based memory controller for fpga based hpc applications, in: High Performance Computing & Simulation (HPCS), 2014 International Conference on, IEEE, 2014, pp. 287–294.

[12] S. Lloyd, M. Gokhale, In-memory data rearrangement for irregular, data-intensive computing, Computer 48 (8) (2015) 18–25.

[13] G. Weisz, J. C. Hoe, Coram++: Supporting data-structure-specific memory interfaces for fpga computing, in: Field Programmable Logic and Applications (FPL), 2015 25th International Conference on, IEEE, 2015, pp. 1–8.

[14] ARM, AMBA AXI and ACE Protocol Specification, Tech. rep. (2011).

[15] Xilinx, LogiCORE IP Product Guide, AXI DataMover v5.1, Tech. rep. (2017).

[16] J. Fjeldtvedt, M. Orlandić, T. A. Johansen, An Efficient Real-Time FPGA Implementation of the CCSDS-123 Compression Standard for Hyperspectral Images, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 11 (10) (2018) 3841–3852.

[17] Naval Research Laboratory, Hyperspectral Imager for the Coastal Ocean (HICO).
URL http://hico.coas.oregonstate.edu/

[18] GICI group, Universitat Autonoma de Barcelona, Emporda software (2011).
URL http://www.gici.uab.es