I.S.C.A. Lab TEI Crete

Data Offloading to FPGA-based Accelerators over PCIe

Authors:
Dimitrios Bakoyiannis
Othon Tomoutzoglou
Georgios Kornaros

Version 1.0 (July 4, 2018)

1 Introduction

This document provides an extended description of a system that combines hardware and software infrastructure for offloading CPU tasks to acceleration units over the PCIe protocol. The aim is to experiment with different approaches of data transfers and memory utilization in order to exploit the system's capabilities for efficient parallel acceleration. The acceleration units of the hardware system implement a Sobel image filtering algorithm and will be referred to as Sobel accelerators.

The system was tested on a host machine with Linux operating system and Intel's Xeon processor. The software part required a user space application and a kernel driver. The hardware part was implemented on the Virtex-7 VC707 FPGA development board (Xilinx) to support communication over the PCIe. The hardware architecture on the FPGA as well as the required custom hardware blocks were designed with Xilinx's Vivado and Vivado HLS tools. Figure 14 provides a detailed overview of the whole system.

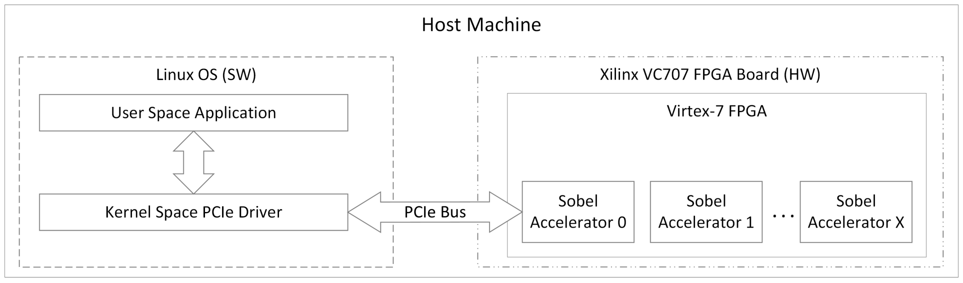

Figure 1: Abstract view of the developed system.

2 Acceleration Approaches

In an abstract scenario, an acceleration procedure requires a user space application, a PCIe kernel driver and the FPGA hardware Sobel accelerator. In such scenario, an image is loaded from the user space application and is transferred through the kernel driver over the PCIe bus to the Sobel accelerator for processing. There are several possible combinations regarding the data movers for transferring the image data and the involved intermediate memories for keeping the pre-processed and post-processed data. In this work, the chosen combinations resulted in three different acceleration approaches that are described in the following subsections:

2.1 Acceleration Direct Approach

In this approach the acceleration procedure requires the usage of kernel memory. A kernel memory allocation is mapped to the user space, thus, the application loads image data to the kernel allocated memory without involving any usage of the user space memory. A data mover that is integrated in the FPGA hardware system transfers the image data from the kernel memory directly to the Sobel accelerator.

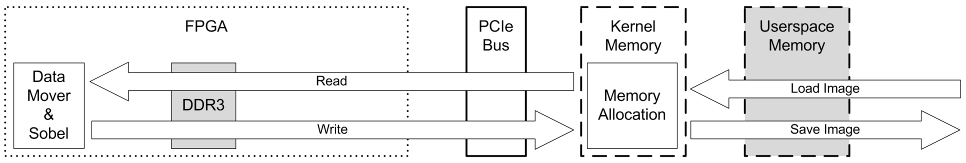

Figure 2: The acceleration Direct approach.

2.2 Acceleration Indirect Approach

In the Indirect approach the acceleration procedure requires the usage of kernel memory as well as the DDR3 memory of the FPGA. Similar to the previous approach, a kernel memory allocation is mapped to the user space, thus, the application loads image data to the kernel allocated memory without involving any usage of the user space memory. The Indirect approach requires two data movers. The first data mover transfers the image data from the kernel memory to the DDR3 memory of the FPGA. The second forwards the image data from the FPGA's DDR3 memory to the Sobel accelerator.

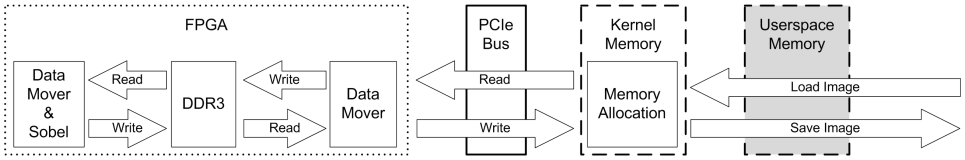

Figure 3: The acceleration Indirect approach.

2.3 Acceleration Scatter/Gather Approach

In the third approach the acceleration procedure uses only user space memory allocations to load the image data. A data mover, in this case, fetches image data from the user space memory to the Sobel accelerator without involving any other data movers or memories. User space memory is allocated in pages without being necessarily contiguous. This particularity requires advanced Scatter/Gather techniques to access the pages that compose the user space memory allocation, thus, the data mover must be aware of the physical address of each page.

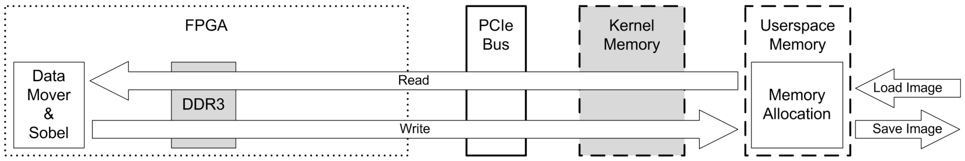

Figure 4: The acceleration Scatter/Gather approach.

3 Hardware Architecture

The current section demonstrates the system's architecture regarding the hardware blocks (IP peripherals) that are part of it and the way that they interact as part of the system. The following subsections give a description of each IP that is integrated in the hardware system.

3.1 Microblaze and Uartlite Controller

The hardware system requires the presence of a Microblaze soft processor for initializing the IP peripherals. Additionally, in collaboration with an interrupt controller the Microblaze can serve interrupt conditions triggered either from a FPGA peripheral or an external source over PCIe.

The Uartlite controller is used by the processor of the system for printing debug messages over the serial protocol (RS-232) to a user terminal.

3.2 PCIe Bridge

The PCIe bridge (AXI Memory Map to PCIe) is a significant controller that acts as a translation bridge for the data transfers between the host's PCIe infrastructure and the FPGA's AXI interface. PCIe packets received from the host operating system are translated to AXI packets and then forwarded to the target peripheral or memory of the FPGA and vice versa.

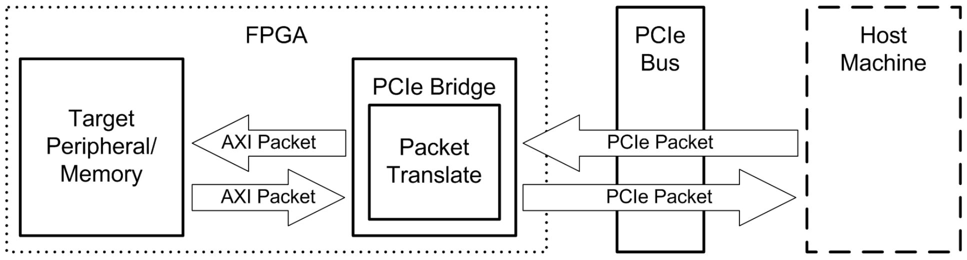

Figure 5: PCIe bridge packets bi-directional translation.

3.3 Data Movers

Data movers are hardware units specialized to make large data transfers from a memory to another. They can be considered as optimal solutions because they can offload a processor from making transfers itself and offer higher transfer rates compared to a processor. The current hardware system integrates two types of data movers:

3.3.1 CDMA Data Mover

The Central Direct Memory Access (CDMA) is the simplest form of data mover. It is used for large data transfers between the kernel memory of the host operating system and the DDR3 memory of the FPGA.

3.3.2 DMA Data Mover

The DMA data mover of this implementation is specialized for streaming large data packets through acceleration units. It is equipped with two AXI master memory map interfaces, one for reading(MM2S channel) and another for writing (S2MM channel) and two AXI stream interfaces, one as an output to the acceleration unit and the other as an input from the acceleration unit. Its operation states are as follows:

1. Read and fetch pre-accelerated data from a source memory to its internal buffer through the MM2S (Memory Map to Stream) AXI master interface.
2. Forward pre-accelerated data through the AXI stream output interface to the target acceleration unit.
3. Receive post-accelerated data through the AXI stream input interface to its internal buffer from the acceleration unit.
4. Write post-accelerated data from the internal buffer to a destination memory through the S2MM (Stream to Memory Map) AXI master interface.

In most cases, where large transactions take place, data are transferred in smaller chunks and the above states are looped until all of them are processed.

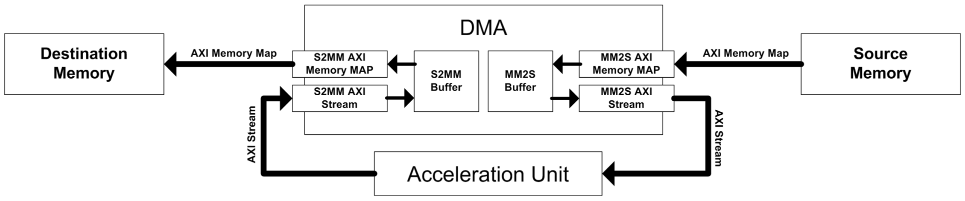

Figure 6: DMA image data transfer procedure.

3.4 Sobel Accelerator

The Sobel Accelerator is the unit responsible for processing image data by applying Sobel edge detection filtering. In order to operate it is paired with a DMA and to comply with the DMA core it requires two AXI stream interfaces as well, one input and one output. It receives image data from the DMA through its stream input interface. It produces one filtered image row once it receives the three first rows. Afterwards, it produces one filtered row for each new row by using the available information from the last three rows buffered internally. The produced image data are forwarded back to the DMA through its output stream interface.

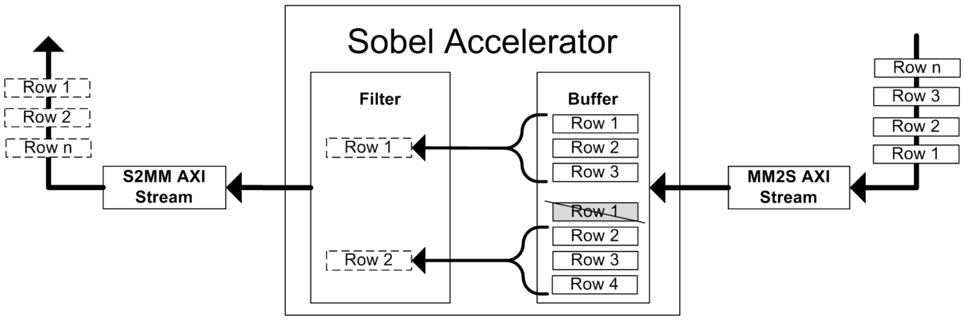

Figure 7: Sobel accelerator image row processing.

3.5 Performance Monitoring

The AXI Performance Monitor unit (APM), in general, is a peripheral that can be used to capture transfer events that take place on AXI interfaces whether it is AXI memory map or AXI stream. Among its several capabilities it can provide information regarding the number of transactions or the number of transferred bytes. Additionally, it supports time measurements in cycles by utilizing its internal global clock counter. The global clock counter is configured as 64 bit and it consists of two 32 bit registers for the MSB and LSB bits of the time value.

The referenced system integrates APMs that are used to monitor the acceleration procedure. One of these APMs is specifically integrated as a means for tracking the parallel acceleration in terms of time placement and duration. The latter is additionally considered as a shared reference timer for the hardware subsystem, the kernel driver and any userspace application. Herein, it will be referred to as Shared Timer. In order to collect the current time value a master peripheral must read both the MSB and LSB registers of the global clock counter.

3.6 FPGA Memories

The hardware system integrates a 256K BRAM (Block RAM) which is accessed through the AXI BRAM Controller. The BRAM is used to store performance metrics and in specific conditions can, also, be used to store information where the acceleration Scatter/Gather approach requires to access scattered pages on the host's memory.

The DDR3 Memory Interface Generator (MIG) is used to establish communication between the 512M DDR3 memory of the FPGA board and the AXI interface of the FPGA system. The DDR3 memory is only required by the acceleration Indirect approach for storing or loading pre-accelerated or post-accelerated image data.

3.7 Acceleration Groups

According to section 2 there are 3 different approaches regarding the involved memories and data movers. That may require special handling on behalf of the hardware system for each approach. Each Sobel accelerator is tightly coupled with specific hardware peripherals that aid the acceleration procedure. These peripherals along with the Sobel accelerator constitute an acceleration group. In order to respect the particularities of each acceleration approach there are three types of acceleration groups:

- Acceleration Group Direct (AGD).
- Acceleration Group Indirect (AGI).
- Acceleration Group Scatter/Gather (AGSG).

Due to constraints that are analyzed in the following sections the hardware system integrates 2 AGDs, 4 AGIs and 1 AGSG.

The peripherals of an acceleration group can be distinguished to those that are instanciated in all the acceleration groups and those that are specialized for a particular type of acceleration group. The peripherals common to all the acceleration groups are:

- Direct Memory Access (DMA).
- AXI Performance Monitor Unit (APM).
- Sobel Accelerator.

The functionality of these peripherals is already described in previous subsections. Nevertheless, it is important to mention the specific usage of the APM as part of an acceleration group. The APM monitors three different AXI interfaces during the four stages described in subsection 3.3.2 where a DMA collaborates with the Sobel Accelerator to process an image. These are:

- The DMA AXI master (MM2S/Read) interface (Slot 0).
- The Sobel AXI stream output interface (Slot 2).
- The DMA AXI master (S2MM/Write) interface (Slot 1).

Comparing the number of transactions and total transferred bytes in each of these interfaces can be vital to ensure integrity and reliability of data transfers during an acceleration procedure. It is, also, a means to gather metrics information for further statistical analysis and a means to watch the behavior of both the DMA and the Sobel Accelerator concerning the way that they manipulate data transfers.

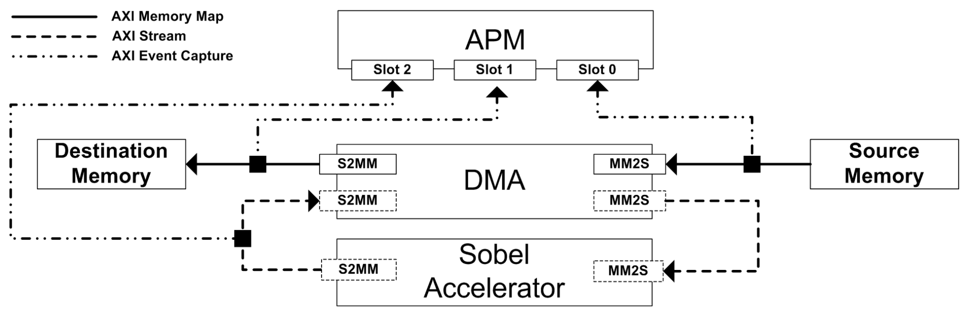

Figure 8: Events capturing with the AXI Performance Monitor Unit (APM).

The peripherals that are available only to a specific acceleration group are specialized for controlling and scheduling the acceleration procedure, thus, they are referred to as schedulers. These are described in the following subsections.

3.7.1 Acceleration Group Direct

The only specialized peripheral for this group is the Acceleration Scheduler Direct core. As its name suggests it controls the whole acceleration procedure with the particularities of the methods that apply in the AGD. The field of activities for this scheduler are:

1. Enable the metric counters and global clock counter of the APM.
2. Initiate the Sobel Accelerator.
3. Initiate DMA transfers.
4. Gather metrics from the APM and the Shared Timer (APM).
5. Trigger an interrupt condition on completion of the acceleration.

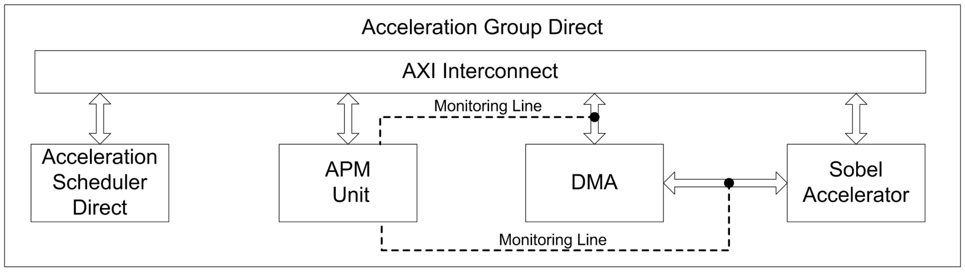

Figure 9: The Acceleration Group Direct (AGD) peripherals.

3.7.2 Acceleration Group Indirect

The basic peripheral specified for this group is the Acceleration Scheduler Indirect core which has similar functionality to the Acceleration Scheduler Direct but controls part of the acceleration procedure. The activities during its operation are:

1. Post acceleration information to the Fetch/Send Info Memory Blocks.
2. Enable the metrics counters and global clock counter of the APM.

3. Initiate the Sobel Accelerator
4. Initiate DMA transfers.
5. Gather metrics from the APM and the Shared Timer (APM).

To aid the AGI acceleration procedure the system requires two CDMAs (Send and Fetch) in order to support concurrent read and write transactions. The CDMA Fetch transfers data from the host's kernel memory to the DDR3 memory of the FPGA while the CDMA Send transfers data vice versa.

The Fetch/Send Info Memory Blocks are a set of registers that retain information required for a CDMA transfer. They, actually, consist of four sets of three registers for the source and destination address and the transfer size in bytes. Each set of registers is dedicated for each of the four AGIs respectively.

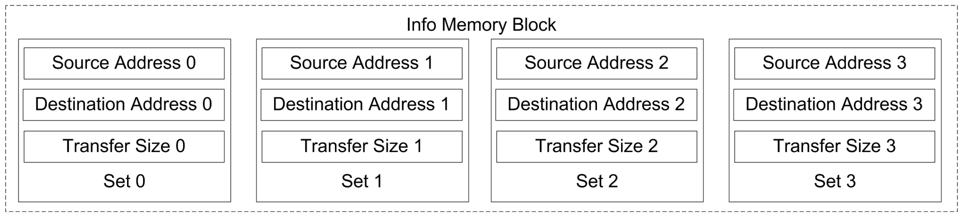

Figure 10: The Info Memory Block.

The AGIs require two additional schedulers which are not part of an acceleration group but control all the AGIs. These are the Fetch and Send Schedulers. The Fetch Scheduler is mainly used to initiate a CDMA transfer according to the information drained from the Fetch Info Memory Block. It checks in Round Robin priority if a AGI has requested a CDMA transfer through its set of registers of the Fetch Info Memory Block. Other responsibilities for that scheduler are:

1. Control the AXI translation registers of the PCIe Bridge.
2. Trigger the Acceleration Scheduler Indirect core on completion of a CDMA (Fetch) transfer.
3. Gather metrics from the Shared Timer.

The Send Scheduler functions almost identically to the Fetch Scheduler. It only differs to triggering an interrupt condition on the completion of a CDMA (Send) transfer which is, also, the completion of the acceleration procedure.

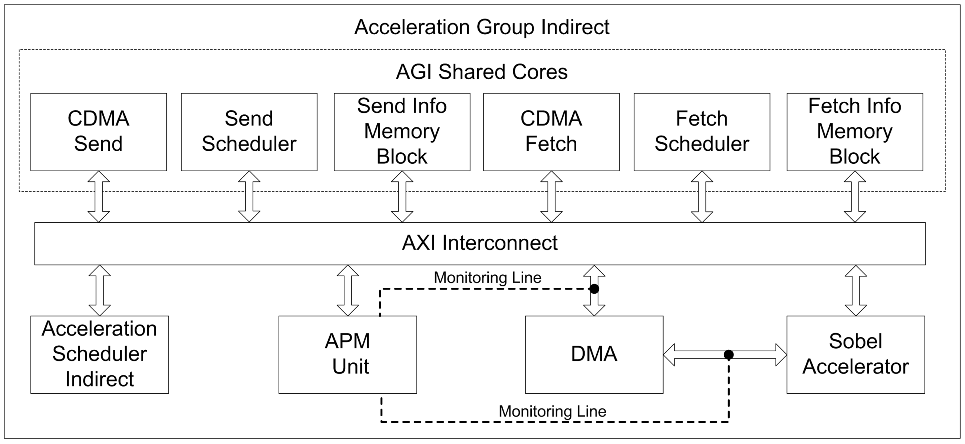

Figure 11: The Acceleration Group Indirect (AGI) peripherals.

3.7.3 Acceleration Group Scatter/Gather

The AGSG is equipped with two schedulers where the leading one is the Acceleration Scheduler Scatter/Gather core that takes over the whole image processing. The second is the DMA Scatter/Gather Scheduler. The controlling activities of the former are:

1. Enable the metrics counters and global clock counter of the APM.
2. Initiate the Sobel Accelerator.
3. Initiate the DMA Scatter/Gather Scheduler.
4. Gather metrics from the APM and the Shared Timer (APM).
5. Trigger an interrupt condition on completion of the acceleration.

The DMA Scatter/Gather Scheduler controls the DMA's MM2S and S2MM channels. This scheduler requires a scatter/gather list with the physical addresses of all the pages that constitute the allocated user space source and destination memories. It reads each physical address from the list and initiates the DMA's MM2S channel to fetch a page each time from the source memory. Similarly, it initiates the S2MM channel to send a page of processed data to the destination memory. The DMA Scatter/Gather Scheduler, also, handles DMA interrupts on completion of each transferred page.

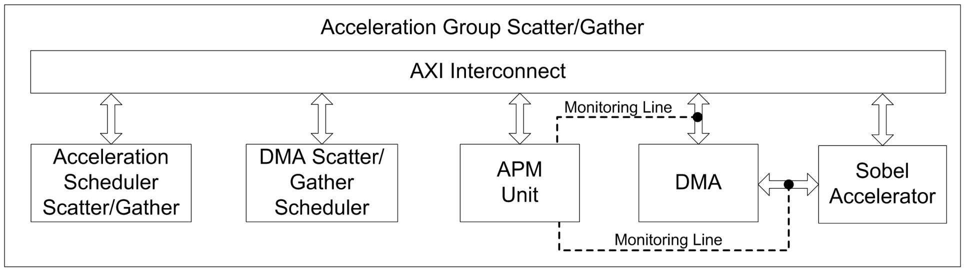
Acceleration Group Scatter/Gather
AXI Interconnect
Monitoring Line
Acceleration Scheduler Scatter/Gather
DMA Scatter/ Gather Scheduler
APM Unit
DMA
Sobel Accelerator
Monitoring Line

Figure 12: The Acceleration Group Scatter/Gather (AGSG) peripherals.

3.8 GPIO Peripherals

GPIO (General Purpose Input/Output) is a type of circuit that provides a number of pins whose behavior, including whether it is an input or output, is controllable by software at run time. The current hardware system integrates for interrupt handling purposes the following three GPIO peripherals:

Table 1: The GPIO integrated peripherals and their attributes.

Peripheral Name	Channel 1 Direction	Channel 2 Direction	Channel 1 Bits	Channel 2 Bits
GPIO PCIe Interrupt	Output	-	32	-
GPIO MSI	Output	Output	1	5
GPIO MSI Read	Input	-	5	-
GPIO ACK	Output	-	1	-

3.9 Interrupt Handling

The hardware part of the system is in tight collaboration with the user space and the kernel space of the host machine. Interrupts and interrupt handling between the host operating system and the FPGA hardware system is integrated within the framework of their cooperation. The hardware system must be able to send interrupts to the kernel driver as well as receive interrupts by the latter.

The kernel driver can be triggered by the hardware system with PCIe MSI interrupts. The PCIe bridge has integrated support for generating MSI interrupts. Their usage makes it possible for a PCIe endpoint device to support sending up to 32 different interrupts according to the vector number of the MSI. Thus, a kernel driver can handle up to 32 different conditions.

The vector number that the PCIe bridge should send as well as the control signal for generating the MSI can be given by an external source. The external source, in this case, is the dual channel GPIO MSI peripheral. Its second channel is a 5-bit output that writes the MSI vector number to the related input of the PCIe bridge while its first channel writes a single bit logic high value for at least one clock cycle that enables the PCIe Bridge to send a new interrupt.

In some cases, though, the Linux kernel does not support multiple MSI handling. In such cases, the vector value written to the GPIO MSI peripheral's second channel output is also written to GPIO MSI Read peripheral input channel. The kernel driver receives a single MSI of zero vector value and the kernel interrupt handler reads the number that is stored at the data register of the GPIO MSI Read core. Then the appropriate routine is called according to the read value.

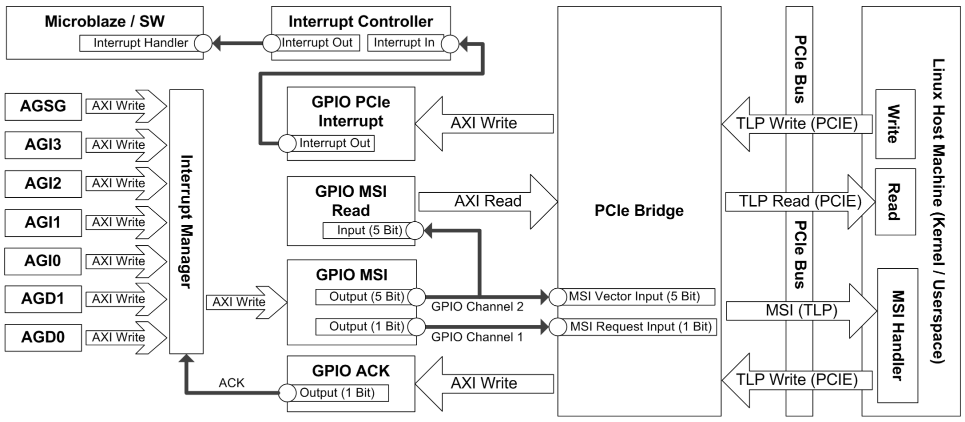

Figure 13: Overview of the system's interrupts.

The kernel driver should be able to receive interrupts by the acceleration groups on completion of their acceleration procedure. Handling interrupt events that may occur simultaneously can be challenging, thus, the hardware system integrates an Interrupt Manager which is a custom hardware block that delivers interrupts in a safe and reliable manner. The Interrupt Manager is equipped with a set of registers which are used to store information regarding the completion of an acceleration process. Each acceleration group writes a non-zero value to a specific register of the Interrupt Manager. The Interrupt Manager checks in round robin its registers for a non-zero value that indicates that an acceleration group has completed its task. Each register of the Interrupt Manager is dedicated to a specific acceleration group. Once a non-zero value is found, the Interrupt Manager writes the corresponding value of the register to the data register of the second channel of the GPIO MSI peripheral which represents a MSI vector number. Afterwards, it writes a logic one value to the data register of the first channel of the GPIO MSI peripheral to generate a new MSI interrupt. The Interrupt Manager remains in polling mode until receiving an acknowledgement signal from the GPIO ACK periperal which indicates the completion of the kernel interrupt handling.

Receiving interrupts by the host system is required in certain cases but the PCIe Bridge does not feature direct support for such demand. An indirect solution is available by integrating the GPIO PCIe Interrupt core. A GPIO peripheral, in general, can be set to trigger interrupts once a new value is written to its data register. As a consequence, making a write transfer from a host's driver or userspace application over the PCIe to the data register of the GPIO PCIe Interrupt core results in providing interrupt support from the host's side. The interrupt line of the mentioned GPIO peripheral is connected to an interrupt controller which informs the Microblaze processor for a new event to handle.

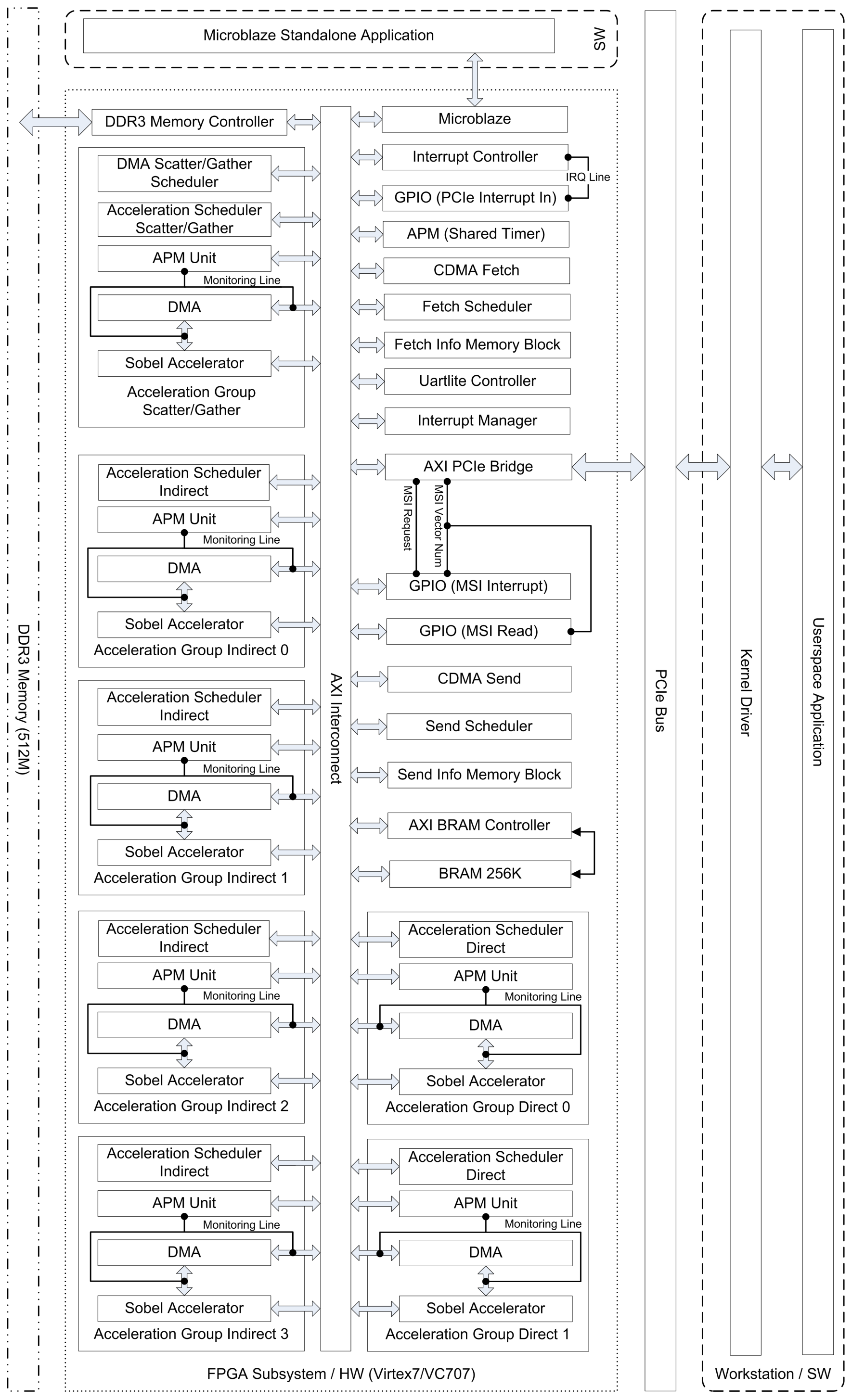

Figure 14: Detailed overview of the hardware architecture.

4 Linux Host and FPGA AXI Address Spaces

The developed system involves heterogeneous architectures that materialize different communication structures, thus, different approaches of interpreting the address space from the perspective of the FPGA platform or the host machine. From the side of the host machine, communication with the hardware platform is achieved through PCIe infrastructure while the FPGA, in native level, uses AXI interface.

The PCIe bridge of the FPGA's hardware system establishes collaboration between the PCIe infrastructure of the host machine and the AXI interface of the FPGA. It is considered as a translation mechanism for transactions between AXI and PCIe packets. Besides translating packets the PCIe bridge, also, exposes the AXI address space to the host machine and the host's address space, whether it is kernel or user space level, to any data mover inside the FPGA.

During translation the PCIe Bridge utilizes its base address registers (BARs) that represent memory windows as seen either by the host system or the FPGA peripherals. The PCIe Bridge is equipped with two types of base address registers AXI BARs and PCIe BARs. AXI BARs make it possible for the FPGA subsystem to access memory areas of the host machine while the PCIe BARs expose to the kernel and user level the address space of the FPGA AXI subsystem which involves off-chip memories and peripherals.

4.1 The AXI BARs of the PCIe Bridge

Each AXI BAR includes one address translation register that can be dynamically assigned with a physical address of the host's memory. Any peripheral that requests access to host's memory areas is unaware of the physical location of that memory in different communication architecture. The AXI BARs are part of the FPGA's native address space, as a consequence, a peripheral that is willing to make read/write transactions from/to the host's memory is actually accessing a AXI BAR base address which is then translated to a physical address of the host machine. This way, any allocated memory in kernel or userspace can become available to the data movers (DMA, CDMA) of the AXI interface by assigning the physical address of that memory to the AXI BAR's address translation register.

The AXI subsystem must be aware of the size of the host's address space that can be accessed through the AXI BAR, thus, a AXI BAR is set to a fixed size of visible address space. The PCIe Bridge supports up to six AXI BARs that can be set to any address space size between 4K to 4G. This way up to six memory areas of the host machine can be exposed simultaneously to the FPGA subsystem. An interesting limitation, in order to ensure reliable data transfers from/to host's memory, is that the defined size of the AXI BAR's address space, which cannot be dynamically configured, must be equal to the size of the allocated memory area on the host's side.

In cases where the FPGA accesses large memory areas in userspace, which are chunked in pages, the subsystem must have perception of the physical address of each of these pages. Exposing all those pages is a matter of reclaiming more complicated scatter/gather techniques. The AXI BARs of the PCIe Bridge must be dynamically assigned with each physical address of each page that a data mover requests to access.

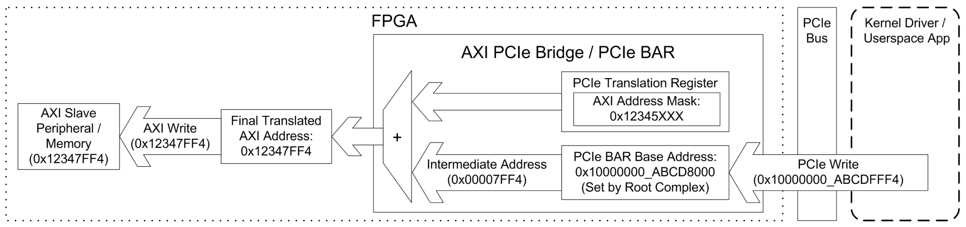
FPGA
AXI PCIe Bridge / PCIe BAR
PCIe Translation Register
AXI Address Mask: 0x12345XXX
AXI Slave Peripheral / Memory (0x12347FF4)
AXI Write (0x12347FF4)
Final Translated AXI Address: 0x12347FF4
+
Intermediate Address (0x00007FF4)
PCIe BAR Base Address: 0x10000000_ABCD8000 (Set by Root Complex)
PCIe Bus
Kernel Driver / Userspace App
PCIe Write (0x10000000_ABCDFFF4)

Figure 15: PCIe Bridge: PCIe BAR address translation.

4.2 The PCIe BARs of the PCIe Bridge

The PCIe BARs, opposite to the AXI BARs, are the means for the kernel driver and any userspace process to claim access over PCIe to off-chip memories and peripherals located inside the FPGA platform.

Each PCIe BAR is configured with a fixed AXI base address and a fixed size of address space that can be visible to the host system. That way the host machine can access any AXI memory or peripheral in the FPGA platform as long as its address space is part of the visible address space of the PCIe BAR. The PCIe bridge includes three 64-bit PCIe BARs that can be configured to represent memory windows from 4K to 4G without any restrictions. A PCIe BAR can be set to make accessible a large amount of address space even if the actual address space inside the FPGA is smaller.

In kernel level, a driver uses functions similar to ioremap in order to map the

FPGA AXI address space as part of the host's virtual address space. Once a kernel driver maps any PCIe BAR to its own virtual address space, the PCIE BARs are, also, exposed automatically to the userspace. A process in the user level can then use the mmap function to map the PCIe BARs as part of the userspace virtual memory.

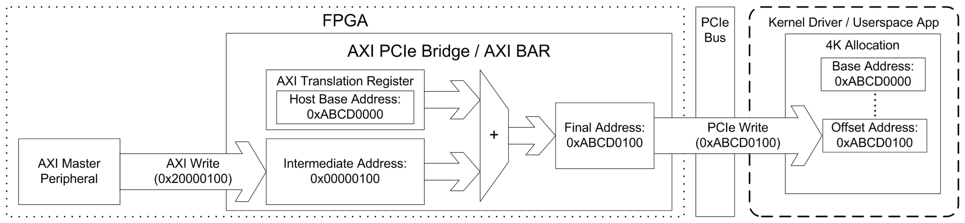
FPGA
AXI PCIe Bridge / AXI BAR
AXI Translation Register
Host Base Address: 0xABCD0000
AXI Master Peripheral
AXI Write (0x20000100)
Intermediate Address: 0x00000100
+
Final Address: 0xABCD0100
PCIe Bus
PCIe Write (0xABCD0100)
Kernel Driver / Userspace App
4K Allocation
Base Address: 0xABCD0000
Offset Address: 0xABCD0100

Figure 16: PCIe Bridge: AXI BAR address translation.

4.3 AXI Address Space Assignment for the PCIe BARs

The AXI address space of the FPGA is designed to be accessed, through the PCIe BARs, both by the kernel driver and any number of userspace applications. The PCIe Bridge is equipped with three PCIe BARs which are assigned with a specific AXI address range in order to make peripherals and memories of the FPGA AXI system part of the virtual address space of the kernel driver or the userspace application. The Linux host machine assigns a virtual base address for each PCIe BAR in order to make the AXI address space that they represent available for reading or writing.

The PCIe BAR0 is set with a 4M memory range starting from 0x10000000 up to 0x103FFFFF of the AXI address space. This range of the AXI address space incorporates all the peripherals of the FPGA system including the seven acceleration groups, the AGI shareable cores and the miscellaneous cores. The base address for each core is set to a specific offset inside the 4M address range. A driver or a userspace application can read/write any register of the FPGA peripheral cores by moving to the desired offset starting from the reserved virtual base address of PCIe BAR0.

The memory range for PCIe BAR1 is set to 256K starting from 0xC0000000 up to 0xC003FFFF. This range represents a 256K AXI BRAM of the FPGA. Similarly, PCIe BAR2 is set to a 512M address range from 0x80000000 to 9FFFFFFF which corresponds to a 512M DDR3 memory on the FPGA board.

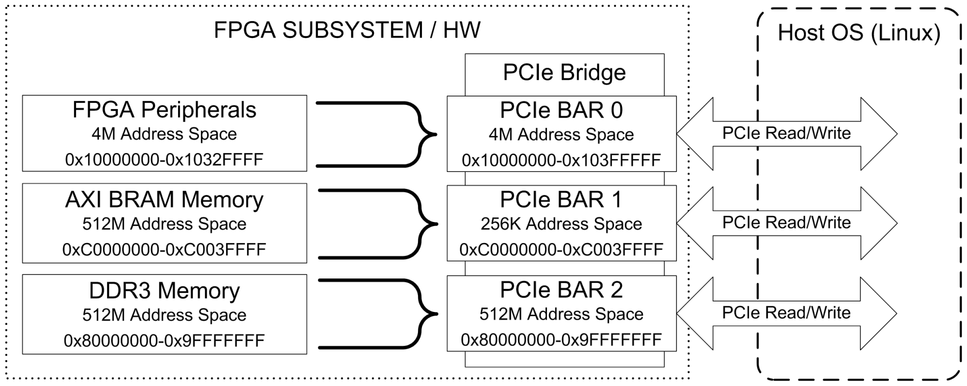

Figure 17: AXI address space assigned for each PCIe BAR.

4.4 Host's Address Space Assignment for the AXI BARs

Any master peripheral of the FPGA is capable of accessing remote memory windows of the host machine's DDR3 memory through the AXI BARs.

Nevertheless, the master peripherals of this implementation that involve accessing the remote memory are those that are part of the acceleration groups. AXI BAR0 is exclusively occupied by the AGD0 (Acceleration Group Direct 0). The DMA of this group uses AXI BAR0 to get direct access to Linux kernel memory allocations where image data are present. AXI BAR1 is, equally, occupied only by the DMA of the AGD1. Both AXI BAR0 and AXI BAR1 are set to a 4M address range.

The DMAs of each AGI (Acceleration Group Indirect) use the DDR3 of the FPGA to read data for processing, in other words, they do not occupy any AXI BAR to get direct access to the host's memory. The role of the CDMA Fetch shareable core is to fetch image data through AXI BAR2 from the host's kernel memory to the offset of the FPGA DDR3 that belongs to the AGI that requested data for acceleration. Similarly, the CDMA Send shareable core will send the accelerated data through AXI BAR3 from the FPGA DDR3 back to the kernel memory. AXI BAR2 and AXI BAR3 are, also, set to a 4M address range. The DMAs of this implementation use two separate AXI channels, one for reading (MM2S) and one for writing (S2MM). The DMAs of the AGDs use one AXI BAR for both channels.

The AGSG (Acceleration Group Scatter/Gather) due to implementation-specific restrictions requires to occupy one AXI BAR for each channel of the DMA. As a result, AXI BAR4 is used by the MM2S channel and AXI BAR5 by the S2MM channel of the AGSG DMA core. AXI BAR4 and AXI BAR5 are set to 4K address range due to accessing memory allocations that are chunked in pages of 4K size.

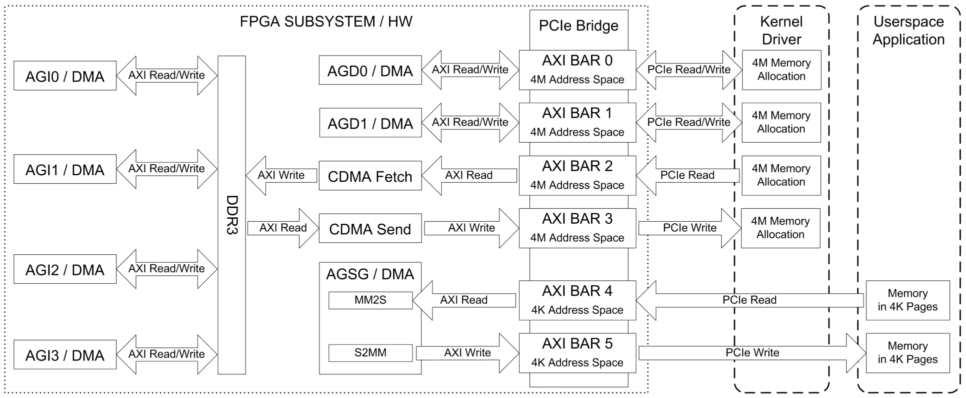

Figure 18: AXI BARs distribution to the FPGA peripherals.

5 Software Architecture

The current section describes the developed software for the needs of the acceleration procedure. The implemented system combines a user space application and a kernel driver to offer a multi-threaded environment that exploits different transfer techniques and resource distribution policies in order to get the maximum gain during data hardware acceleration.

5.1 Linux Kernel Space Driver

The kernel driver is the intermediate level for the communication between the user space application and the FPGA hardware system over the PCIe infrastructure.

It detects the PCIe BARs of the FPGA's PCIe bridge and maps accordingly the AXI address space of the hardware system as part of the kernel's virtual address space. As a result, the peripherals and memories of the FPGA can be accessed by the driver with simple read/write operations.

In the current implementation, the userspace application can offload image acceleration tasks through the kernel driver. The latter is capable of receiving system calls (ioctl or input/output control) from the user space which are preset commands that force the kernel driver to execute specified routines.

Among other operations, the user space application can use the ioctl calls to request occupation of an available hardware accelerator. One responsibility of the driver on receiving such request calls is to manage and distribute the hardware acceleration resources to the user space threads.

The kernel driver, also, manages the user space threads regarding their requests for occupying acceleration units. The available accelerators are assigned to user space threads with respect to certain distribution policies. The threads that cannot be served are set to sleep mode until at least one accelerator is available.

In certain cases, the user space application requests to have direct access to kernel memory fragments. In such cases, the kernel driver allocates specified amount of cache coherent kernel memory and maps it as part of the user virtual address space.

In other cases, the AGSG of the FPGA requires direct access to memory allocations in user space level. The particularity of the user space is that memory can be allocated as a collection of 4K sized pages which are not necessarily in contiguous memory. On request, the kernel driver uses scatter/gather techniques to create a list of physical addresses of the pages that synthesize the user space memory allocation. The AGSG uses this list to transfer data directly from/to the userspace memory.

Last but not least, the kernel driver handles MSI interrupts received on completion of an acceleration task by the acceleration groups through the PCIe bus. The driver collects metrics regarding the whole acceleration procedure and triggers the respective user space thread that requested the current acceleration.

5.2 Linux User Space Application

The user space application is a multi-threaded environment that offloads image processing tasks to accelerators. The hardware architecture as well as the number of the accelerators or their availability is abstract to the application.

The application creates a desired number of threads according to a user's request. Each thread commands the kernel driver to acquire access to kernel memory allocations in order to load image data for processing. Afterwards, threads request from the driver to ocupy acceleration resources of the FPGA.

Each thread remains in polling mode by reading a specified flag which on changing its state indicates the completion of a requested acceleration procedure. The threads collect metrics regarding the acceleration process that are saved in a common XML file for further analysis.

5.3 The Microblaze Soft Processor of the FPGA

The Xilinx's software development kit (XSDK) offers driver libraries which are automatically generated according to the present peripherals of the hardware design. The software developed for the Microblaze soft processor utilizes the appropriate driver functions to initialize and setup the peripherals of the hardware system.

In addition, the Microblaze initializes its interrupt controller and registers certain function routines in order to receive and handle interrupt events.

6 Kernel Resource Distribution Policies

In section 5 it is mentioned that the user space application is not aware of the hardware system and the available accelerators. The driver, on the other hand, is in tight collaboration with the hardware system and controls the acceleration groups on behalf of the user space threads.

An acceleration group cannot be assigned to a thread without the permission of the kernel driver which is aware of the population of the acceleration groups as well as their state. In this project, the driver applies the $Greedy_{Single}$ and $Greedy_{Partition}$ policies in order to distribute the available acceleration resources.

6.1 The Greedy Single Ditribution Policy

The scheduling principle of the $Greedy_{Single}$ policy is to strictly assign a single accelerator to a user-space process that requested image acceleration. For each new acceleration request the driver must choose for the most effective among all the acceleration groups of the FPGA system (seven in our platform), where the effectiveness metric is the inverse of the latency cost for an acceleration task. The AGDs are proved to require the lower completion time and the AGSG the higher one. The driver checks to find the first available acceleration group in the following order:

$AGD0 \rightarrow AGD1 \rightarrow AGI0 \rightarrow AGI1 \rightarrow AGI2 \rightarrow AGI3 \rightarrow AGSG$

The state, occupied or available, for every acceleration group, is monitored by maintaining status flags inside the driver. If the driver finds an available group, then, it changes the state of its corresponding flag to occupied and activates the acceleration group. The flag will only change its state to available inside an interrupt routine, which is triggered on completion of the acceleration procedure. This mechanism ensures that the acceleration group will be explicitly locked to a userspace process until the latter is served.

6.2 The Greedy Partition Distribution Policy

Given the capacity of the accelerator groups to parallelize not only discrete userspace jobs, but also to parallelize a single job, the goal of the $Greedy_{Partition}$ policy is to occupy as many acceleration groups as possible on a user request for image processing. The image in this case, will be splitted in equal parts and accelerated by more than one Sobel processing units in a concurrent manner. In the $Greedy_{Partition}$ policy, the driver searches and assigns all the acceleration groups that are found available.

On a new request, the driver searches to occupy the largest number of available acceleration groups for a single process, according to their state flags following the same order as in the $Greedy_{Single}$ policy. However, in the $Greedy_{Partition}$ strategy only AGD and AGI groups are considered to process workloads that are partitioned. AGSG requires special handling due to using userspace memory allocation, which memory is chunked in pages and thus, it is only occupied if no other acceleration group is found available.

The driver, similarly, changes the status flags for each available group to occupied and starts the acceleration groups after they are given information regarding the size and the offset in memory where the image partition, which they are going to process, is located. Each acceleration group that processes its assigned image partition triggers an interrupt routine inside the driver, which changes the state of its flag to available. The process is considered served after all the corresponding occupied groups become available.

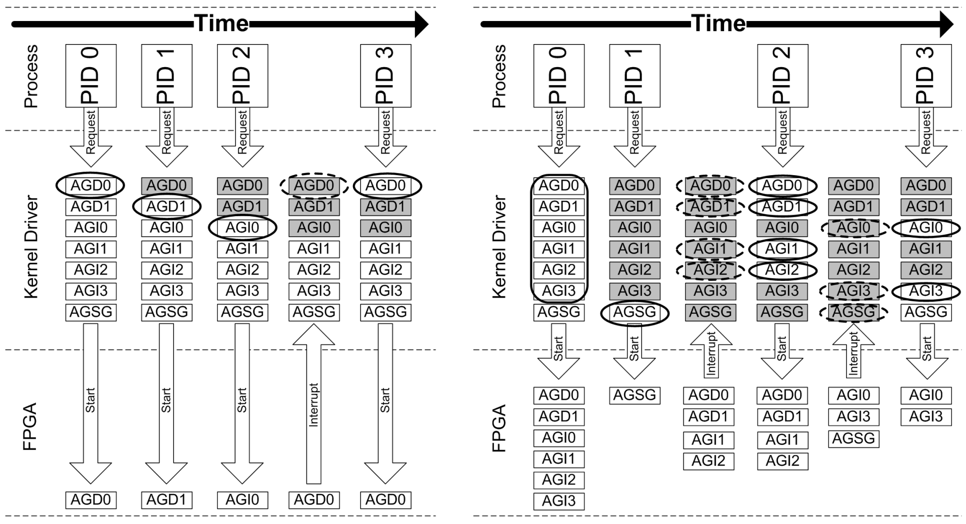

Figure 19: Examples of the $Greedy_{Single}$ (Left) and the $Greedy_{Partition}$ (Right) policies.

7 Acceleration Proceedings

The objective of the developed system is to provide off-chip hardware acceleration facilities to HPC machines. The particular interest in this project is to ensure arbitration between multi-threaded processes of a host machine that require to occupy the available resources of the FPGA subsystem.

The AGD is intended to optimize acceleration performance by diminishing memory utilization and by providing the acceleration unit with direct access to the pre- processed image data. AGD makes memory allocations only in kernel space. Userspace applications use kernel memories to load image data or to get accelerated images. The DMA core in this group can have direct access to the kernel memories through an AXI BAR which should be explicitly occupied for one AGD. Explicit use of AXI BARs for concurrent memory accesses leads to a restricted number of AGDs since the PCIe Bridge can have up to six available AXI BARs.

The AGI requires both kernel memory allocations and the DDR3 of the FPGA board. A userspace application loads image data directly to the kernel memory which are, then, copied by a CDMA core to the DDR3 memory of the board through an AXI BAR. The DMA unit of a AGI uses the DDR3 memory to read and process data. Accelerated data are written back to DDR3 and then forwarded by another CDMA core through a second AXI BAR to the kernel memory. AGIs, compared with the AGDs, have performance limitations due to making extra data copies between memories but it is optimized to require only two shareable AXI BARs for any number of AGIs.

The AGSG is a solution which only involves memory allocations in userspace for storing pre-processed and post-processed image data. In this group the DMA is optimized to access directly the userspace memory. Memory allocations in userspace, though, are a collection of pages that are scattered throughout the userspace memory which can impact the overall performance.

This section of the paper is intended to describe the flow where different hardware and software components synchronize and collaborate to serve processes that require Sobel edge detection to image data by utilizing special hardware acceleration units. The AGD, AGI and AGSG types of acceleration groups differ in memory usage and transfer techniques. The following subsections will give an extended description of the acceleration procedure through the collaboration of the hardware and software components for each acceleration group. Each subsection describes part of the acceleration procedure in sequential steps.

7.1 Thread Preparation and Acceleration Group Request

The starting point for the acceleration procedure is the userspace where a thread of the application is intended to request image data acceleration.

The initial action, which is referred to as the preparation procedure, is to allocate memory resources. The thread uses system calls to the driver which is dictated to allocate memory buffers in the kernel space and exposes those memories to the userspace by utilizing the mmap() function. This method reduces memory usage since a userspace process can have direct access to kernel space memories. Each thread has its dedicated set of kernel memory allocations which are freed after the thread terminates its execution.

During the preparation procedure the process creates a chunk of memory where the pre-processed image data are loaded, another chunk of memory to keep the accelerated (post-processed) image and a third chunk of memory to keep metrics and time measurement information. The size of the allocated buffers is strictly set to 4M in order to comply with the AXI BAR address space limitations that are discussed to a previous section of the current document which are, also, set to 4M size. It is important to mention that the preparation procedure takes place only once when the process is started and the allocated memories are being reused as many times as required.

The next step for the thread is to copy the pure image data to its dedicated pre-process memory area of the kernel space. It is important to mention that the initial parent thread reads the image file (.bmp) and loads the pure image data (not including the file headers) to a shared user space memory allocation that is commonly accessed by all child threads. This technique was used to reduce the latency cost of loading an image file multiple times.

Afterwards, the user space thread uses a system call (IOCtl) to request from the driver to occupy any available acceleration resources. Depending on the distribution policy the driver will try to occupy any of the AGD or AGI resources. If none of them is found available it will check if the AGSG is available. In case the status of the AGSG indicates availability the user space thread will be informed through a flag to make a new request to obtain access to the AGSG.

7.2 Assigning Acceleration Resources

The kernel is set by the programmer to apply whether the $Greedy_{Single}$ or the $Greedy_{Partition}$ policy. When the driver receives a new request for acceleration by a thread it checks for available acceleration groups which will be locked to the thread according to the applied resource distribution policy (section 6).

The driver uses flags for each acceleration group that indicate whether they are available or occupied. If all the acceleration groups are occupied the user space thread is set to sleep mode until at least one is found available otherwise the driver proceeds to assign hardware resources to the thread. The user space thread reads a specific flag in polling mode until the flag indicates the completion of the acceleration.

If the applied policy is the $Greedy_{Single}$ the driver will lock only one acceleration group respecting the priorities referred to subsection 6.1.

In the case where the $Greedy_{Partition}$ policy is used the driver must lock to the current thread all the available acceleration groups that belong to the AGD or AGI types. If AGSG is also found available it is locked for the current thread as a worst case scenario only if all AGDs and AGIs are occupied. Since the $Greedy_{Partition}$ policy may require multiple acceleration groups the driver separates the image data to segments equal to the number of the occupied acceleration groups.

7.3 Initiate the Assigned Acceleration Groups

In this step the driver has already locked a single or multiple hardware accelerators according to the distribution policy.

Each acceleration group that is assigned to a thread requires certain information regarding the acceleration procedure that must be provided by the kernel driver. When multiple acceleration groups are assigned to a thread the driver must provide to each acceleration group information about the specific image data segment that each is going to process. This step of the acceleration procedure can be considered as the initiation of the acceleration groups.

The driver can access the registers of the FPGA peripherals through the PCIe BAR0. As a result, the initiation of each acceleration group can be accomplished directly from the driver. The following subsections describe the particularities when initiating the AGDs, AGIs and AGSG.

7.3.1 Initiate the AGD

The first step to initiate an AGD is to setup a AXI BAR of the PCIe bridge with the physical address of the kernel allocated memory where the image data are located. The driver writes the physical address to the translation register of the corresponding AXI BAR of the PCIe bridge. This action is required so that the DMA of the AGD can access the image data of the kernel memory through the AXI BAR. The AXI BAR0 is utilized by AGD0 while the AXI BAR1 is utilized by AGD1.

The driver must also provide the Acceleration Scheduler Direct peripheral of the AGD with the source address of the pre-processed image data as well as the destination address where the processed data should be written. The driver sets both as source and destination the address of a common AXI BAR. Another approach would be to occupy one AXI BAR for reading the pre-processed image data from the kernel memory and a second AXI BAR for writing the processed data. In this implementation each AGD occupies one AXI BAR due to their restricted availability. As a result, the pre-processed and the post-processed data utilize the same kernel memory allocation.

The Acceleration Scheduler Direct requires information regarding the resolution of the image that will be processed. The driver accesses specific registers of the AGD scheduler to write the number of rows and columns of that image.

The last step for the driver is to enable the Acceleration Scheduler Direct by writing a start command to the control register of the latter. The scheduler takes over the whole acceleration procedure and interrupts the kernel driver on completion of the data processing.

7.3.2 Initiate the AGI

The procedure for initiating the AGI is similar to the initiation of the AGD. The driver must provide to the AGI the source and destination addresses of the kernel memory. These addresses are written to the Acceleration Scheduler Indirect of the AGI instead of setting directly tha AXI BARs.

The driver enables the Acceleration Scheduler Indirect after writing the number of rows and columns of the image or image segment to the registers of the latter. The Acceleration Scheduler Indirect takes over the acceleration procedure in collaboration with the Fetch and Send Schedulers.

7.3.3 Initiate the AGSG

The initiation of the AGSG requires more complicated handling. A user space thread that is assigned with the AGSG must allocate user space memories both for the source and destination image data.

The particularity of these memory allocations is that they are chunked in 4K sized pages. The AGSG must be aware of the location of each page that must access. As a result, the thread requests from the driver to create a scatter/gather list of all the pages that constitute the source and destination user space memory allocations. In other words, these lists include the physical addresses of all the allocated pages.

Once the source and destination scatter/gather lists are created the user space thread makes a final request to the driver for occupying the AGSG. The driver writes the two lists to the FPGA's BRAM through the PCIe BAR1.

Lastly, the driver provides the number of rows and columns of the image to the Acceleration Scheduler Scatter/Gather and then enables the latter similarly to the previous schedulers by writing a start command to its control register. The Acceleration Scheduler Scatter/Gather handles the acceleration proceedings of the AGSG.

7.4 Hardware Proceedings

This subsection is intended to describe the followed procedure once the driver has initiated an acceleration group. The control unit of a group is the acceleration scheduler. The actions of a scheduler during the acceleration procedure differ depending the acceleration approach.

7.4.1 The AGD Acceleration Procedure

The Acceleration Scheduler Direct of the AGD carries information, which is provided by the kernel driver, regarding the data source and destination memories as well as the image resolution. The following sequential steps describe the acceleration process when controlled by the Acceleration Scheduler Direct:

a. The Acceleration Scheduler Direct accesses the control register of the AXI Performance Monitor (APM) peripheral of the AGD in order to enable its counters. The counters of the APM provide information about the number of AXI transactions and the number of the transferred bytes.

b. Read the current time value from the Shared Timer in order to get the moment that the acceleration procedure started.

c. Initiate the Sobel Accelerator so that it can receive and process image data. The Acceleration Scheduler Direct provides the Sobel Acelerator with information regarding the image rows and columns and then enables the latter by accessing its control register.

d. Initiate the DMA engine to transfer the image data. This step requires to enable the MM2S and S2MM channels. The S2MM channel is enabled first in order to be ready for receiving the first processed data. The functionality of the DMA is described in subsection 3.3.2. The Acceleration Scheduler Direct writes the destination physical address to the destination register of the S2MM channel and then accesses the control register of the latter to enable it. The scheduler, then writes the transfer size to the DMA's S2MM channel which initiates the S2MM transfer. Similarly, the scheduler writes the source physical address and the transfer size to the registers of the MM2S channel in order to start the MM2S transfer.

e. The Acceleration Scheduler Direct remains in polling mode until receiving an interrupt from the S2MM channel of the DMA which indicates that the data are processed and written to the destination memory.

f. Read the current time value from the Shared Timer in order to get the moment that the acceleration procedure completed.

g. The scheduler accesses the control register of the AXI Performance Monitor (APM) peripheral of the AGD to disable its counters in order to avoid metrics capture of unwanted events.

h. Access the status register of the S2MM channel of the DMA in order to acknowledge the triggered interrupt.

i. Gather the metrics that are captured by the counters of the APM. These metrics are presented in the following table:

Table 2: The captured events of the AGD.

Event Name	DMA Channel
Read Transactions	AXI MM2S
Write Transactions	AXI S2MM
Read Bytes	AXI MM2S
Write Bytes	AXI S2MM
Stream Packets	AXI Stream
Stream Bytes	AXI Stream
GCC Low	-
GCC High	-

j. Update the value of the control register of the APM in order to reset its counters including its global clock counter.

k. Trigger an interrupt condition on completion of the acceleration. The scheduler writes a MSI vector number which corresponds to the current Acceleration Group to a specific register of the Interrupt Manager. The latter will forward the interrupt request by generating a MSI interrupt in collaboration with the PCIe bridge.

7.4.2 The AGI Acceleration Procedure

The Acceleration Scheduler Indirect of the AGI similarly to the AGD carries information, which is provided by the kernel driver, regarding the data source and destination memories as well as the image resolution. The sequential steps below describe the acceleration process when controlled by the Acceleration Scheduler Indirect which is accomplished in collaboration with the Fetch and Send Schedulers:

Acceleration Scheduler Indirect - Initiate the Fetch Scheduler

a. The Acceleration Scheduler Indirect initially accesses a specific set of registers of the Fetch Info Memory Block which is dedicated to the current AGI in order to write the source and destination addresses, a possible offset where data are located and the size of the upcoming data transfer. This information will be used by the Fetch Scheduler for initiating the CDMA Fetch peripheral to make the transfer from the kernel source memory to the FPGA's DDR3 memory.

b. The Acceleration Scheduler Indirect remains in polling mode until receiving a signal from the Fetch Scheduler on completion of the CDMA Fetch transfer.

Fetch Scheduler - The CDMA Fetch Data Transfer

a. The Fetch Scheduler keeps reading each of the 4 sets of registers of the Fetch Info Memory Block until it is aware of a new data transfer on behalf of an AGI. At this step it is aware that the current AGI required a new data transfer.

b. The Fetch Scheduler accesses and updates the control register of the CDMA Fetch peripheral in order to enable the latter's interrupt capabilities.

c. The CDMA Fetch is used to read data from the kernel memory through a dedicated AXI BAR of the PCIe bridge. At this step the Fetch Scheduler reads the source physical address of the kernel memory from the Fetch Info Memory Block and assigns the dedicated AXI BAR with it. The next step is to set the CDMA's source address register with the base address of the AXI BAR. As a result, the CDMA Fetch will read data from the base address of the AXI BAR which is actually translated by the PCIe bridge to read transactions from the source kernel memory.

d. The Fetch Scheduler reads the destination physical address, which is an offset of the DDR3 memory of the FPGA, from the current set of registers of the Fetch Info Memory Block and writes it to the destination address register of the CDMA Fetch peripheral.

e. The Fetch Scheduler reads the current time value from the Shared Timer in order to get the moment that the CDMA Fetch transfer started.

f. The Fetch Scheduler writes the transfer size that was found at the Fetch Info Memory Block to the bytes to transfer (BTT) register of the CDMA Fetch. This action initiates the data transfer of the CDMA Fetch peripheral.

g. The Fetch Scheduler remains in polling state until it receives an interrupt by the CDMA on completion of the transfer.

h. Once the CDMA transfer is finished the Fetch Scheduler reads the counters of the Shared Timer to get the completion time.

i. The Fetch Scheduler accesses the status register of the CDMA Fetch peripheral in order to acknowledge the triggered interrupt and then updates the value of the CDMA Fetch control register so as to reset the latter and re-enable its interrupts.

j. The Fetch Scheduler clears the current set of registers of the Fetch Info Memory Block so that it can be aware of a new transfer request of the corresponding AGI.

k. Finally, the Fetch Scheduler triggers a signal which informs the current Acceleration Scheduler Indirect for the completion of the CDMA Fetch transfer.

Acceleration Scheduler Indirect - The Acceleration Procedure

a. At this point the Acceleration Scheduler Indirect exits from the polling mode and starts the acceleration of the image data which are now located at the DDR3 of the FPGA. The steps of the acceleration are exactly the same as those that are described in steps a. to j. of the 7.4.1 subsection

Acceleration Scheduler Indirect - Initiate the Fetch Scheduler

a. On completion of the acceleration, the Acceleration Scheduler Indirect should write to the corresponding set of registers of the Send Info Memory Block the source and destination addresses, a possible offset where the processed data are located and the size of the upcoming data transfer. This information will be used by the Send Scheduler for initiating the CDMA Send peripheral in order to transfer the processed image data from the FPGA's DDR3 memory to the kernel destination memory.

Send Scheduler - The CDMA Send Data Transfer

a. The Send Scheduler, similarly to the Fetch Scheduler, keeps reading each of the 4 sets of registers of the Send Info Memory Block until it is aware of a new data transfer on behalf of an AGI. At this step it is aware that the current Acceleration Scheduler Indirect required a new data transfer.

b. The Send Scheduler accesses and updates the control register of the CDMA Send peripheral in order to enable the latter's interrupt capabilities.

c. The Send Scheduler reads the source physical address, which is an offset of the DDR3 memory of the FPGA, from the current set of registers of the Send Info Memory Block and writes it to the source address register of the CDMA Send peripheral.

d. The CDMA Send is used to write data to the kernel memory through a dedicated AXI BAR of the PCIe bridge. At this step the Send Scheduler reads the destination physical address of the kernel memory from the Send Info Memory Block and assigns the dedicated AXI BAR with it. The next step is to set the CDMA's destination address register with the base address of the AXI BAR. As a result, the CDMA Send will write data to the base address of the AXI BAR which is actually translated by the PCIe bridge to write transactions to the destination kernel memory.

e. The Send Scheduler reads the current time value from the Shared Timer in order to get the moment that the CDMA Send transfer started.

f. The Send Scheduler writes the transfer size that was found at the Send Info Memory Block to the bytes to transfer (BTT) register of the CDMA Send. This action initiates the data transfer of the CDMA Send peripheral.

g. The Send Scheduler remains in polling state until it receives an interrupt by the CDMA Send on completion of the transfer.

h. Once the CDMA transfer is finished the Send Scheduler reads the counters of the Shared Timer to get the completion time.

i. The Send Scheduler accesses the status register of the CDMA Send peripheral in order to acknowledge the triggered interrupt and then updates the value of the CDMA Send control register so as to reset the latter and re-enable its interrupts.

j. The Send Scheduler clears the current set of registers of the Send Info Memory Block so that it can be aware of a new transfer request of the corresponding AGI.

k. Finally, the Send Scheduler should trigger an interrupt condition on completion of the CDMA Send transfer which indicates that the acceleration is completed and the processed image data are written to the destination kernel memory. The scheduler writes a MSI vector number which corresponds to the current Acceleration Group to a specific register of the Interrupt Manager. The latter will forward the interrupt request by generating a MSI interrupt in collaboration with the PCIe bridge.

7.4.3 The AGSG Acceleration Procedure

The AGSG was developed for handling an acceleration procedure under more complicated conditions where the source and destination memories of the image data are chunked in pages of the user space. In order to read or write data the AGSG requires a scatter/gather list of the physical addresses of all the pages that constitute the user space memories.

The next steps describe the acceleration procedure of the AGSG which controlled by the Acceleration Scheduler Scatter/Gather and the DMA Scatter/Gather Scheduler. The former is responsible to accomplish the whole acceleration procedure while the latter must control both the MM2S and S2MM channels of the DMA in order to read the image data and write the processed image data page by page.

Acceleration Scheduler Scatter/Gather - Initiate the AGSG

a. The Acceleration Scheduler Scatter/Gather accesses the control register of the APM of the AGSG in order to enable its counters.

b. The next step is to read the global clock counter value of the Shared Timer so as to get the time that the acceleration started.

c. Initiate the Sobel Accelerator so that it can receive and process image data. The Acceleration Scheduler Scatter/Gather, similarly to the other schedulers, provides the Sobel Acelerator with information regarding the image rows and columns and then enables the latter by accessing its control register.

d. Enable the interrupts of the DMA Scatter/Gather Scheduler.

e. The Acceleration Scheduler Scatter/Gather writes the transfer size to the registers of the DMA Scatter/Gather Scheduler and then accesses the latter's control register in order to enable it.

f. At this point the Acceleration Scheduler Scatter/Gather remains in polling mode until the DMA Scatter/Gather Scheduler triggers an interrupt on completion of the data transfer which is, also, the completion of the acceleration.

DMA Scatter/Gather Scheduler - Control the DMA Transfer

a. The DMA Scatter/Gather Scheduler starts by calculating the number of pages that must be transferred according to the transfer size.

b. The next step for the DMA Scatter/Gather Scheduler is to access the control registers of the MM2S and S2MM channels in order to enable their interrupts.

c. The DMA Scatter/Gather Scheduler reads the physical address of the first page from the source scatter/gather list and assigns it to a source AXI BAR that is dedicated for accessing the source memory of the user space.

d. The DMA Scatter/Gather Scheduler sets the source address register of the MM2S channel of the DMA with the base address of the source AXI BAR.

e. Then, the DMA Scatter/Gather Scheduler enables the MM2S channel by updating the value of its control register and writes the transfer size to its bytes to transfer (BTT) register in order to initiate a transfer. The tranfer will be of size equal to the page size or smaller if there are remaining bytes.

f. Similarly to the MM2S channel, the DMA Scatter/Gather Scheduler assigns the destination AXI BAR with the first page of the destination scatter/gather list and then enables the S2MM channel to transfer the chunk of processed data to the user space destination memory.

g. The DMA Scatter/Gather Scheduler remains in polling mode until it receives an interrupt either from the MM2S or the S2MM channel on completion of the page tranfer.

h. When an interrupt occurs the DMA Scatter/Gather Scheduler acknowledges the interrupt event of the current channel by updating the value of its status register.

i. The next step is to check for more pages to transfer for the current channel and enable if necessary a new page transfer similarly to the previous steps.

j. Once all pages are tranferred for both channels, the DMA Scatter/Gather Scheduler triggers an interrupt which targets the Acceleration Scheduler Scatter/Gather.

Acceleration Scheduler Scatter/Gather - Complete the AGSG

a. Once the Acceleration Scheduler Scatter/Gather receives an interrupt from the DMA Scatter/Gather Scheduler it exits its polling mode and reads the global clock counter of the Shared Timer in order to get the Time that the acceleration completed.

b. The scheduler accesses the control register of the AXI Performance Monitor (APM) peripheral of the AGSG to disable its counters in order to avoid metrics capture of unwanted events.

c. The Acceleration Scheduler Scatter/Gather accesses the interrupt status register (ISR), the interupt enable register (IER) and the global interrupt enable register (GIE) of the DMA Scatter/Gather Scheduler in order to clear and re-enable the interrupts of the latter.

d. Gather the metrics that are captured by the counters of the APM. These metrics are the same that are presented in table 3 of the AGD.

e. Update the value of the control register of the APM in order to reset its counters including its global clock counter.

f. Trigger an interrupt condition on completion of the acceleration. The Acceleration Scheduler Scatter/Gather writes a MSI vector number which corresponds to the AGSG to a specific register of the Interrupt Manager. The latter will forward the interrupt request by generating a MSI interrupt in collaboration with the PCIe bridge.

7.5 Acceleration Interrupts and Kernel Interrupt Handling

The previous steps on subsection 7.4 describe the acceleration procedure of each acceleration group once initiated by the kernel driver. As already described, each acceleration group on completion informs the Interrupt Manager in order to generate an interrupt.

The Interrupt Manager includes 7 registers where each one is dedicated to one acceleration group. An acceleration group writes a request value to its dedicated register of the Interrupt Manager. The request value is actually a MSI vector number which is associated to the acceleration group. The Interrupt Manager reads the 7 registers in round robin policy for a new interrupt request and generates a MSI by writing the vector number of the current register to the second channel of the GPIO MSI peripheral. Then the Interrupt Manager writes a logic one value to the first channel of the GPIO MSI peripheral. The outputs of the two channels of the GPIO peripheral are connected to the PCIe bridge and force the latter to generate a MSI interrupt.

The kernel driver is registered with 7 interrupt handlers for each of the 7 acceleration groups. Once receiving an MSI interrupt the kernel runs the interrupt handler that is associated to the corresponding acceleration group that requested the interrupt.

The interrupt handler collects all the metrics that where captured in the kernel driver and the acceleration group and then updates a flag that indicates that the acceleration group is no longer occupied.

The user space thread that reads that flag exits its polling mode since now it is aware that the acceleration procedure is completed.

The kernel driver writes a logic one value to the GPIO ACK peripheral which is connected to the Interrupt Manager. The Interrupt Manager receives the acknowledgement signal from the GPIO ACK which indicates that the current MSI interrupt is handled so the Interrupt Manager can check for triggering a new one.

Table 3: The MSI vector number for each acceleration group and their kernel interrupt handlers

Acceleration Group	MSI Vector	Kernel Interrupt Handler
AGD 0	0	Handler 0
AGD 1	1	Handler 1
AGI 0	2	Handler 2
AGI 1	3	Handler 3
AGI 2	4	Handler 4
AGI 3	5	Handler 5
AGSG	6	Handler 6

7.6 The Last Steps on Completion of the Acceleration

As mentioned in subsection 7.5 a user space thread exits its polling state once the expecting MSI interrupt is received.

The user space thread must save the processed image data to a new file. The processed image data is located in the destination kernel memory if the thread occupied any of the AGDs or the AGIs or in the user space destination memory if occupied the AGSG.

The final step is to save all the metrics that are collected by the user space thread, the kernel driver and the acceleration group. All metrics of any threads that were activated and occupied acceleration groups are stored in XML form in a common XML file.