Energy-Performance Considerations for Data Offloading to FPGA-based Accelerators over PCIe

DIMITRIOS MBAKOYIANNIS, OTHON TOMOUTZOGLOU, and GEORGE KORNAROS,
Technological Educational Institute of Crete, Greece

Modern data centers increasingly employ FPGA-based heterogeneous acceleration platforms as a result of their great potential for continued performance and energy efficiency. Today, FPGAs provide more hardware parallelism than is possible with GPUs or CPUs, while C-like programming environments facilitate shorter development time, even close to software cycles. In this work we address limitations and overheads in access and transfer of data to accelerators over common CPU-Accelerator interconnects such as PCIe. We present three different FPGA Accelerator dispatching methods for streaming applications (e.g., multimedia, vision computing). The first uses zero-copy data transfers and on-chip scratchpad memory for energy efficiency, the second uses also zero-copy but shared copy engines among different accelerator instances and local external memory. The third uses the processor's MMU to acquire the physical address of user pages and uses scatter-gather data transfers with scratchpad memory. Even though all techniques exhibit advantages in terms of scalability and relieve the processor from control overheads through using integrated schedulers, the first method presents the best energy efficient acceleration in streaming applications.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; **Heterogeneous (hybrid) systems**; • **Hardware** → *Reconfigurable logic and FPGAs*; *Hardware accelerators*;

Additional Key Words and Phrases: PCIe-based FPGA accelerators, zero-copy streaming accelerators, adaptive dispatching, scatter-gather DMA

ACM Reference Format:
Dimitrios Mbakoyiannis, Othon Tomoutzoglou, and George Kornaros. 2017. Energy-Performance Considerations for Data Offloading to FPGA-based Accelerators over PCIe. *ACM Transactions on Architecture and Code Optimization* 1, 1, Article 1 (December 2017), 25 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

A progressive landscape in architecture and software emerges with an increasing trend to move beyond homogeneous parallelism. Accelerators complement CPU cores to enhance single thread performance with tightly coupled accelerators or complement multi-core performance with loosely coupled accelerators through PCIe, Infiniband or QPI attach. In addition, in data center ecosystems, accelerators not only can process local workloads, but can also be shared over the network among many servers and their workloads. To achieve better efficiency in terms of power and execution time, datacenter servers utilize accelerators whether

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Association for Computing Machinery.
XXXX-XXXX/2017/12-ART1 $15.00
https://doi.org/0000001.0000001

GPUs, FPGAs, or specialized-function ASICs (e.g. Google's Tensor Processing Unit (TPU)). Developing costs and ease of programming are not in favor of the last option. At the same time, both classes of devices, GPUs and FPGAs, can support abundant parallelism, as both have hundreds to thousands of arithmetic units available on each chip. Fully-programmable accelerators (e.g. GPUs) excel in delivering far better performance than traditional cores in deep learning and in many scientific applications by leveraging the single-instruction (or thread), multiple-data execution model. However, reconfigurable architectures like FPGAs offer solutions that consolidate higher computing and energy efficiency. In this direction High Performance Computing (HPC) systems integrate reconfigurable fabric into their computing nodes to accelerate datacenter applications [1][2][10][26]. Many challenges are raised for FPGAs in this domain, since, first, the datacenter servers often host multiple applications, each of which may need a separate accelerator. Second, multiple objectives can be intertwined over time, ranging from server-side energy efficiency, to applications short latency and to handling fine-grain or coarse-grain workloads. Moreover, several datacenter applications need tighter monitoring of execution and differentiation of service levels, which requirements push towards access to accelerators internal monitoring and to enabling dispatching hints in the applications.

Significant effort has been spent in optimizing the communication process between CPU and FPGA accelerators. Traditionally, the most widely used integration method is to connect FPGA accelerators to CPU via PCIe with accelerators equipped with local, private memory. To illustrate the context, Figure 1 outlines a typical heterogeneous, accelerator-rich system. The accelerator-enhanced (ACC) architecture includes one or more processor cores loosely coupled with many hardware accelerators. Each accelerator may include Scratch Pad Memory (SPM) or private external Local Memory (LM). Direct Memory Access (DMA) controllers commonly are responsible to realize the streaming data communication between the processor(s) and the accelerators. The Catapult fabric [24] attacks PCIe communication

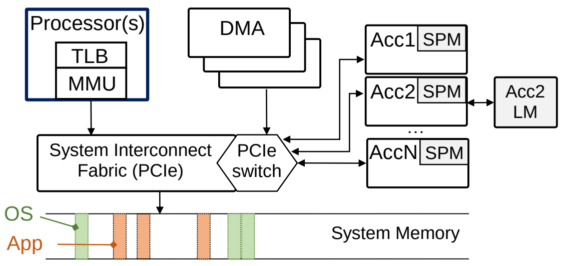


Fig. 1. Architecture of a typical heterogeneous system extended with PCIe-connected accelerators.

latency by avoiding system calls and by modifying servers memory organization to allocate communication buffers in non-paged, user-level memory. Other industrial systems that provide FPGA-based supercomputing nodes to accelerate specific application domains (e.g., data mining workloads by Convey HC-1 [2], or financial applications by Maxeler technologies [13]), minimize data transfer latencies to off-chip accelerators by using compute-communication overlapping with coarse-grain workloads.

Additionally, direct memory access (DMA) engines commonly transfer application data from the "slow" CPU path, to the accelerator domain-realm to process them using their customized hardware engines, and return the results back to the CPU memory system or, for further processing by other accelerators or servers. The major challenge for all dominant programming frameworks in datacenter servers that have been streamlined is to minimize the movement of data [23][24]. For applications that live in user space, an offload process requires copying data via the Operating System (OS) to/from the accelerator and manually maintaining data consistency. The value of accelerator-enabled servers is essentially weighed against the cost of the extra data copies that they introduce. Instead of maintaining two copies of data in both host and device address spaces, zero-copy methods, via OS support, have been proposed, or, unifying virtual addressing between the host CPU and accelerators. However, the latter requires hardware support by I/O memory management units (IOMMUs), which incur overheads that are non-trivial to handle; latency of servicing a TLB miss is significantly higher on a GPU than on a CPU ($\sim$25% [29]). In addition, to eliminate the data-transfer overhead for loosely-coupled accelerators (i.e., PCIe-connected GPUs or FPGAs), software solutions suggest to attain high data-transfer bandwidth using pinned memory [28], and overlap computations with data transfers [9]. Commonly, the underlying PCIe bus favors bandwidth-oriented applications instead of applications with low-latency and fine-grained interactions. Further, to provide contiguous memory buffers to hardware accelerators via using the kernel's Contiguous Memory Allocation (CMA) is hard to operate with no concerns [20]. Short- or long-lived pinned pages and CMA itself that provides physically contiguous pages at runtime without exclusive reserved memory area are partial and mostly application-specific solutions.

To address these challenges the major contributions of this work are as follows.

- We optimize dispatching through avoiding user-kernel copies by using direct access to contiguous kernel-space memory or to user-space memory without requiring support by an I/O Memory Management Unit (IOMMU)
- We optimize interrupt management and synchronization via offloading the control functions from the host to hardware custom schedulers.
- We enable support for programmable dispatching, matching the type and size of workload and the servers needs for energy efficiency to the appropriate type of hardware-threaded accelerators over PCIe. Three different accelerator types are designed featuring: (i) data processing directly from the host CPU memory, (ii) using a discrete external DDR memory shared by many accelerator cores, and (iii) scatter-gather DMA to access directly user-space non-contiguous data. All methods are free from user- kernel-bounce buffer copies.
- We optimize PCIe FPGA-side AXI Base Address Registers (BAR) utilization and host-accelerator address mapping with the goal to support (i) energy efficient acceleration through powering-off accelerators local DDR memory, (ii) pipelining workload transfers with accelerators computations in the case that the DMA throughput exceeds the accelerators processing throughput, and (iii) acceleration for over-sized workloads, which apparently cannot be managed in contiguous memory space in the host and require processing in scatter-gather fashion.

The remainder of this paper is structured as follows. First, in Section 2 we give an overview of related work. Section 3 presents the design of key components of the proposed infrastructure, Section 4 describes the policies developed in the kernel driver to dynamically assign jobs to the accelerators. Section 5 analyzes the specific hardware and software components of our infrastructure that allow dynamic monitoring of the accelerators and collection of metrics, which are embedded in an analysis and visualization tool. Section 6, based on a real design, reports on our evaluation in terms of performance, energy and scaling capacity, and section 7 concludes our work.

2 RELATED WORK

Much effort has been invested in integration techniques for accelerator-rich systems examining tightly coupled accelerators (Teimouri et al., target to increase accelerators utilization as they scale [27]) and loosely coupled accelerators, even with varying levels of memory hierarchy [4]. All studies though are simulation-based instead of using actual platforms. IBM's CAPI [26] architecture enables the FPGA to participate as a cache coherent peer to CPU without any software intervention and corresponding latency, which is perfect for workloads that need frequent communication with general-purpose cores; however, the CAPI protocol requires CPU awareness (Power8 CPU) and thus we do not further consider this solution for our framework.

Recent studies investigate traditional PCIe-based CPU-FPGA platforms and the latest quick-path interconnect (QPI) based Intel-Altera HARP platform with coherent shared memory [3]. HARP brings tighter accelerator integration to the processor via a unified address space among them, which enables zero-copying using pinned host memory. However, the programmer must still use special application programming interfaces (APIs), similar to PCIe type accelerators, to allocate pinned memory space. While Intel HARP favors fine-grained interaction, PCIe-based platforms can catch up with a QPI-based platform if the designer uses batch-processing to hide the latency of computation kernel invocation overhead [3]. Moreover, with higher throughput of evolving PCIe standard and several reconfigurable architectures that are increasingly developed for the acceleration of cloud computing applications in data centers [14], our framework brings important optimization in efficient energy-performance use of FPGA accelerators via PCIe.

Past works in PCIe frameworks present solutions mainly to optimize utilization of PCIe throughput. JetStream [30] and RIFFA 2.1 [11] are capable of reaching 97% of the maximum achievable PCIe link utilization during transfers. RIFFA 2.1[11] uses PCIe to connect FPGAs to CPU's system bus while supporting more classes of Xilinx FPGAs, multiple FPGAs in a system, more PCIe link configurations, higher bandwidth, and Linux and Windows operating systems. User-level control though, dynamic mapping of host-accelerator domains, interrupting and adaptation of transferring data to accelerators are hard to develop or not supported at all, in favor of simplifying the software API. The EPEE library [7] allows flexible host-FPGA PCIe communication and developers can access kernel buffers directly, similar to part of our proposed work, which eliminates the memory copy operations between kernel and user buffers. In SAccO[31], the portable and scalable accelerators are implemented on standard FPGA boards connected via PCIe for accelerating parts of multiprocess streaming applications[31], where a high-level communication API is provided which uses the same function calls for SW-SW communication via sockets and SW-HW communication via PCIe. Instead, we address the design of adaptive hardware and software mechanisms around PCIe communication bridging and we demonstrate actual application-accelerator improved communication, bringing insight to scaling while adapting to either energy or performance. JetStream [30] focuses on optimizing throughput over PCIe for third generation PCIe and on multi-FPGA hosted accelerators. However, it lacks support for integrating multi-accelerators per FPGA device; this solution assumes hardware accelerator that is able to use the full PCIe gen3 throughput, which is not always the case. Integration of user logic inside the FPGA requires extra effort than using the simple PCIe bridge IP core wrapper provided by Xilinx [32], as in our framework. Further, the JetStream implements single-, double-buffering for source and destination data and zero copy methods, while our solution realizes additionally hardware-controlled scatter-gather DMA transfers directly from-, to- user space and considers the optimal utilization of AXI BAR resources among different on a single board accelerators. In addition in our framework, hardware schedulers integrated with each accelerator offload the task to monitor data transfers and acceleration progress from the kernel driver and thus they offer higher performance and energy efficiency. We are very confident that our framework can easily support multiple FPGAs per host.

Finally, only ffLink [5] and our framework notably considers buffer-size constraints in the OS. The Contiguous Memory Allocation (CMA) in the Linux kernel can provide contiguous buffers on request, but is limited to a maximum size of 4 MB per allocation, which in turn limits the maximum size of a single DMA transfer to 4 MB. To the best of our knowledge our framework is the only one to consider memory management on the FPGA side in concert with memory management in the host side. Table 1 gives an overview of distinctive features of PCIe-based infrastructures.

Table 1. Summary of PCI express-based frameworks for FPGA accelerators

	RIFFA 2.1	EPEE	ffLink	JetStream	This work
scatter-gather DMA	✓		✓	✓	✓
zero copy (mmap,copy-to-FPGA)	✓	✓	✓	✓	✓
zero copy (mmap,no-FPGA-copy)					✓
dispatch policies					✓
integral accelerator monitoring					✓
hw controlled dispatching					✓
support for multiple accelerators					✓
easy user-logic interfacing		✓	✓		✓

While modern architectures are equipped with IOMMUs [15] to provide address translation support[1] for loosely coupled devices, including customized hardware accelerators and GPUs, these IOMMUs introduce extra overheads, in many cases, in terms of complexity and power/performance, due to many accesses to page tables in external system memory and the synchronization required [29] [19]. Even though IOMMUs offer hardware-based operation with caching (significantly faster of course than software methods [17]) and even shared second-level TLB to improve performance [8], our proposed architecture proves that the IOMMUs are essentially unnecessary in all of our methods. First, for user memory that can be non-contiguous, if we leverage the host CPU's MMU to perform the virtual to physical translations and we pin the pages to memory, then our DMA-extended design inside the scatter-gather type of accelerator transparently provides the physical space to the accelerator. Second, by directly placing the data that an offloading process needs inside kernel's contiguous memory (zero-copy), provides the accelerator with IOMMU-free access to host's physical memory.

3 ACCELERATOR SYSTEM ARCHITECTURE

The aforementioned challenges call for the development of specialized dispatching mechanisms to enable parallelism, high-throughput data transfers with zero overhead, differentiated service levels, and finally, a homogeneous unified API at the programming level. In addition,

[1] An IOMMU also applies a sand-boxing mechanism to protect the system against illegal memory accesses; however, the processor's MMU provides better protection[19]

GPGPU computations seem to fade when considering FPGAs, especially for various compositions of mathematical operations that are computable in streaming data flows, such as FFT, vision computing and recent trend of deep neural networks [21]. Hence, we opted for FPGA-based accelerators for image filtering acceleration via PCIe interconnect.

We selected an application domain that can clearly benefit from customized hardware processing engines, that is, Sobel edge detection of images. Although it is difficult to formalize and apply improvements from domain-specific solutions to the broader field of accelerator-extended system architecture, limiting the impact of such work, we believe that our proposed dispatching methods and cores can provide significant performance and energy improvements over a wide range of applications that are amenable to hardware acceleration. Temporal and spatial access locality of the workloads along with a high computation to data movement ratio can lead to high-performance acceleration. However, as analysed in Shao et al.[25], applications with large memory stride, such as fft can barely see acceleration benefits. Next, we consider workloads that exhibit streaming behavior.

To address different perspectives in terms of efficiency, namely energy, latency, performance and system restrictions such as non-contiguous user address space, we developed three different communication methods to stream data to and from the Sobel hardware accelerator. Beyond designing accelerators that are efficient while addressing multiple objectives, i.e., performance, low energy consumption, optimal memory management, it is equally important to exploit the potential of increased parallelism of FPGA devices (potentially matching the PCIe available bandwidth). However, by simply tiling instances of the Sobel hardware accelerator delivers poor scaling and negligible performance gains due to hardware constraints. For example, the Xilinx PCIe bridge core integrates six AXI base address registers (BARs) for translation between the different physical address space domains, for *upstream* transfers, initiated by the IP cores inside the FPGA. Moreover, chip-sets have limits to the amount of system resources they can allocate and the PCIe bridge core must be configured to adhere to these limitations. Hence, we developed particular hardware components that integrate specialized units in addition to the Sobel hardware accelerator, to monitor and manage the acceleration procedure. Thus, we hereby call as *acceleration group* the main Sobel accelerator wrapped with these control-management units. Depending, mostly, on the methodology that is used to transfer image data, we developed three optimized acceleration groups.

- Acceleration Group Direct (AGD): data are fetched directly from host's kernel-space memory, processed and send back in a streaming fashion with zero copying, nor from/to user-/kernel- memory, or from/to the FPGAs local DDR memory.
- Acceleration Group Indirect (AGI): data are initially copied from host's kernel-space memory to FPGAs local DDR memory, then streamed to the accelerator using the local copy and finally data are transferred to the host's memory.
- Acceleration Group Scatter/Gather (AGSG): data are fetched directly from host's user-space physical memory in chunks of size equal to host's page size; first a scatter-gather list is prepared with the physical base addresses of the source pages that contain the source image data and of the target pages that will store the processed image data; then, the acceleratorconsults this list to process the data.

AGD and AGI accelerators require a userspace application to initially invoke an ioctl system call to request for contiguous memory space and thus avoid the costly copies of image data from user- to kernel-space that a traditional driver would perform. AGD is the most energy-efficient and high-performance method since it utilizes only on-chip buffering. Moreover, AGD is the most power efficient solution since it performs zero copies and it requires zero off-chip memory space; notice that the power consumed in the Virtex7 device I/O memory interface reaches up to 1.663W with 50% read / 50% write cycles at 800 Mb/s (26% of total device power consumption) as measured by the Vivado Power Estimator tool. However, each AGD requires one AXI base-address translation register (BAR) in the PCIe bridge, reserved during the acceleration process. Interestingly, PCIe endpoint bridge by Xilinx requires the address space window defined in one AXI BAR to match the size of the kernel space allocated by the CMA. Therefore, we optimize the dispatching methods to initially perform the allocation of the kernel space and then to dynamically program the corresponding AXI BAR region. Thus, to enable truly parallel AGD type accelerators, a dedicated AXI BAR is binded per accelerator.

On the other hand, the AGI solution allows decoupling of reserving the PCIe bridge base address translation registers (BARs) from the actual acceleration process; hence, multiple Sobel image accelerators can run in parallel working on their local memory data copies that reside in the FPGA off-chip DDR3 memory, while new applications transfer new images in this local DDR3 memory. Each AGD accelerator requires a dedicated AXI BAR for read and writes, while many AGI Sobel accelerators share two AGI AXI BARs through a CDMA IP core. Finally, similar to AGI, AGSG uses one AXI BAR for reads and another AXI BAR for writes, while only AGSG offers dynamically hardware-controlled AXI BAR mappings. Table 2 summarizes PCI bridge BAR usage.

Table 2. BAR assignment for integrated Acceleration Groups. BAR dynamic programming is done by the kernel driver, except the AGSG BARs 4 and 5 that are programmed on the fly by the AGSG circuitry

PCIe Bridge BARs [a]	Operation	Assignment
AXI BAR 0	Read/Write	AGD0
AXI BAR 1	Read/Write	AGD1
AXI BAR 2	Read	CDMA fetch for all AGIs
AXI BAR 3	Write	CDMA send for all AGIs
AXI BAR 4	Read	AGSG
AXI BAR 5	Write	AGSG
PCIe BAR 0	Read/Write	4MB region: peripherals of Acc. groups
PCIe BAR 1	Read/Write	256KB region: metrics BRAM
PCIe BAR 2	Read/Write	512MB region: DDR3 FPGA memory

[a]Base Address Registers.

The hardware constraints (i.e., FPGA capacity, PCIe throughput) and the requirements for differentiated dispatching steered to the development of a multi-objective accelerator through integrating two AGD, four AGI and one AGSG hardware accelerator groups in a Xilinx Virtex7 device (VC707 platform). Table 3 lists the involved hardware cores for each acceleration group.

The acceleration groups are complemented with miscellaneous IP cores to facilitate efficient interrupt handling and accelerator system monitoring. The functionality of these cores is briefly presented next.

- The AXI Memory Map to PCIe (PCIe Bridge) [32] is an important controller that acts as a protocol and address translation bridge for the data transfers between the host's PCIe infrastructure and the FPGA's AXI4-interconnected components. The AXI4

Table 3. Basic building cores of each Acceleration Group

IP Cores	AGD	AGI	AGSG
Sobel Accelerator (custom function)	✓	✓	✓
DMA	✓	✓	✓
AXI Performance Monitor Unit (APM)	✓	✓	✓
Acceleration Scheduler Direct	✓		
Acceleration Scheduler Indirect		✓	
Acceleration Scheduler Scatter/Gather			✓
DMA Scatter/Gather			✓

Memory Mapped to PCI Express core translates the AXI4 memory read or writes to PCIe Transaction Layer Packets (TLP) packets using up to six 64-bit Base Address Registers (BARs) and translates PCIe memory read and write request TLP packets to AXI4 interface commands using up to three 64-bit Base Address Registers (BARs). Please notice that it is compliant only to PCIe second generation.

- The Microblaze soft-processor is mainly used to initialize the acceleration groups and the rest of the peripheral cores. Together with an Interrupt Controller it serves interrupts triggered either by a FPGA peripheral or by an external source over PCIe.
- The GPIO peripherals are utilized to provide interrupt support over the PCIe Bridge. These GPIO peripherals are: (i) GPIO PCIe Interrupt In, (ii) GPIO MSI and (iii) GPIO MSI Read.

We developed and integrated a specific mechanism in our infrastructure to tackle deficiencies of the PCIe bridge and of the Linux kernel to provide interrupt support in both directions, to/from the accelerators. The use of MSI interrupts makes it possible for a PCIe endpoint device to support sending up to 32 different interrupts on the basis of the vector number of the MSI. Thus, the kernel driver must handle up to 32 different conditions. In some cases, though, the Linux kernel does not support multiple MSI handling (e.g. Linaro Stable Kernel (LSK) for ARM/ARM64 within microserver environments). In such cases, the vector value written to the GPIO MSI Upstream peripheral's second channel output is also copied 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 kernel invokes the appropriate routine according to this value. When the acceleration groups complete a task, they generate interrupts and write a vector value in dedicated registers in the Interrupt Manager. This block checks in a round robin manner for completed jobs and generates the corresponding MSI through the GPIO MSI peripheral. The Interrupt Manager triggers MSI interrupts to the kernel driver ensuring zero loss of interrupts.The Interrupt Manager guarantees elimination of synchronization side-effects of MSI interrupts between the bridge and kernel-side interrupt handling when concurrent acceleration completions take place.

Table 4 gives an overview of the prominent properties of each acceleration group. Hence, instead of focusing in increasing the performance of an FPGA accelerator, via intra-parallelism, or focusing in adding extra lanes or moving to more advanced PCIe gen version, we focus in optimizing the objectives shown in the three rightmost columns in the following way; we opted for establishing a combination of all the attributes instead of using a baseline based in either of these groups and attempting to optimize this baseline. Note that we have not considered partial reconfiguration, which is among our future plans.

Table 4. Summary of accelerators key attributes and performance, energy and scalability impact

Attributes \Accelerators, Objectives	AGD	AGI	AGSG	Perf	Energy	Scalability
Use of on-chip scratchpad memory (private)	✓		✓	✓	✓	
Use of external local memory (shared)		✓				✓
Data zero-copy host memory (mmap, host MMU)	✓	✓	✓	✓	✓	✓
Data zero-copy host-Acc local memory	✓		✓	✓	✓	
PCIe Bridge AXI BAR (shared)		✓			✓	✓
Dataset size limitation-free (4MB)			✓			✓

3.1 Acceleration Groups

All three categories of acceleration groups include a Direct Memory Access (DMA) block, the main Sobel Accelerator and an AXI Performance Monitor Unit (APM). A DMA engine is an optimized hardware solution that offloads a processor by making large data transfers from one memory location to another. The DMA of this implementation is specialized for streaming large data packets through acceleration units. As shown in Figure 2 left, it is equipped with two AXI master memory-map interfaces, one for reading and another for writing and two AXI stream interfaces, one as an output and the other as an input. Its standard functions involve:

(1) Read and fetch input data from a source memory to its internal buffer through the MM2S (Memory Map to Stream) AXI master interface.
(2) Forward input data through the AXI stream output interface to the target acceleration unit.
(3) Receive output data through the AXI stream input interface to its internal buffer from the acceleration unit.
(4) Write output 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 repeated until all data are processed.

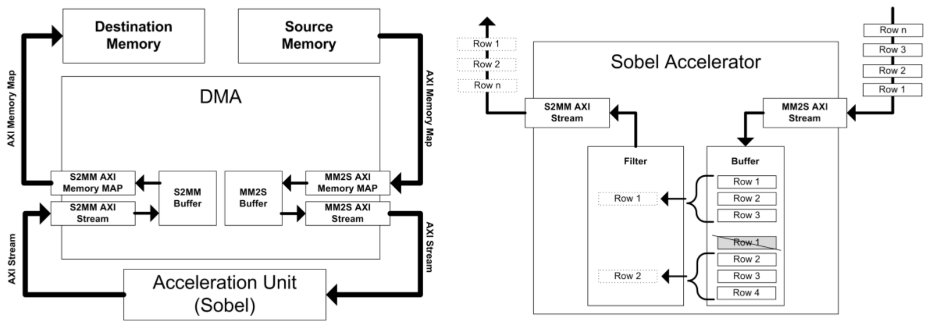


Fig. 2. (a) DMA layout in support of streaming in and out data to an accelerator, (b) Internal Sobel accelerator data flow.

The *Sobel Accelerator*, as shown in Figure 2 right, is the unit responsible for processing image data by applying Sobel edge detection filtering. It is paired with a DMA core using two AXI stream interfaces to fetch and send data. Once the Sobel core receives the three first rows of the image data from the DMA it produces one row of the filtered image. Afterwards, it produces one filtered row by using the available data from the three more recent rows stored in its internal buffer.

The *AXI Performance Monitor Unit* (APM) monitors the DMA AXI master (MM2S/Read) interface, the Sobel AXI Stream output interface and the DMA AXI master (S2MM/Write) interface. Monitoring the number of transactions and the total transferred bytes in each of these interfaces can be vital to ensure integrity and reliability of data transfers during an acceleration procedure, while it is also a means to gather statistical metrics.

3.2 AGD Accelerator

The key ideas in the design of accelerators include first the separation of the computation function from the overall dispatching method. Thus, although we developed an image filter accelerator, essentially streaming type exhibiting data parallelism are appicable as well. Second, a key principle is to include the control plane of the accelerator, so as to free the processor for useful work.

The AGD component, as shown in Figure 3, includes a customized controller which we hereby call *Acceleration Scheduler Direct* core. This core controls the acceleration process after the corresponding kernel driver signals a new job. The controller-scheduler initiates the following sequence of operations: (i) it enables the metrics counters and the global clock counter of the APM, (ii) it initiates the Sobel Accelerator, (iii) it initiates the DMA transfers and monitors the DMA copying completion, (iv) it gathers metrics from the APM and the Shared Timer (APM), and, (iv) it informs the Interrupt Manager peripheral to generate MSI interrupts.

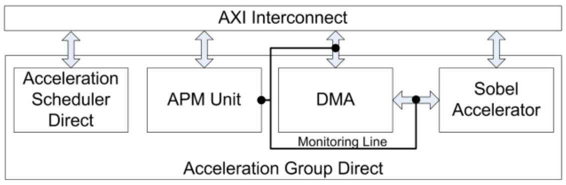


Fig. 3. The Acceleration Group Direct (AGD) block-level organization.

3.3 AGI Accelerator

Figure 4 shows the AGI component, which also integrates a specialized controller, hereby named *Acceleration Scheduler Indirect* core, which controls part of the acceleration procedure. The activities during its operation involve: (i) it initializes acceleration information in the Fetch/Send Info Memory Blocks, (ii) it enables the metrics counters and global clock counter of the APM, (iii) it initiates the Sobel Accelerator, (iv) it initiates DMA transfers, (v) it gathers metrics from the APM and the Shared Timer (APM).

The Central Direct Memory Access (CDMA) is a hardware unit specialized to make large data transfers from one memory location to another using AXI burst transfers. The AGI acceleration procedure requires two CDMAs (Send and Fetch) to support concurrent read and write transactions and on top, can programme the AXI BARs independently.

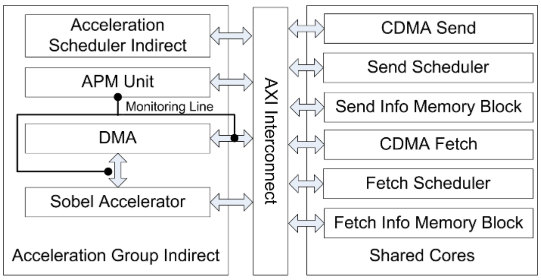


Fig. 4. The Acceleration Group Indirect (AGI) Cores; scaling this accelerator requires multiple instantiating of the AGI component only, the set of cores on the left of the AXI interconnect. The CDMA and its companion cores coordinate the transfers between the host and the local memory.

The Fetch/Send Info Memory Block consist of four sets of three registers, each set stores the source and destination address and the transfer size in bytes. Each set of registers is dedicated to each of the four AGIs respectively. The Fetch Scheduler checks in round-robin priority if an AGI has requested a CDMA transfer through its set of registers inside the Fetch Info Memory Block and initiates a CDMA transfer. In addition, this scheduler, (i) controls the AXI translation registers of the PCIe Bridge, (ii) triggers the Acceleration Scheduler Indirect core on completion of a CDMA (Fetch) transfer, and (iii) gathers metrics from the Shared Timer (APM). The Send Scheduler functions almost identically to the Fetch Scheduler. It only differs in that it informs the Interrupt Manager peripheral to generate an MSI interrupt after the completion of a CDMA (Send) transfer; this completion also signals the completion of a corresponding AGI acceleration procedure.

3.4 AGSG Accelerator

The AGSG component is designed with the key principle to handle input and output data organized in pages in user-space memory, and essentially without the need of an I/O Memory Management Unit, yet providing the accelerator access to user's virtual address space.

The AGSG is equipped with two different schedulers, where the primary one is the Acceleration Scheduler Scatter/Gather core that controls the whole acceleration process (see Figure 5), which means it starts the Sobel, starts the DMA scheduler and monitors the process. The secondary one is the DMA Scatter/Gather Scheduler which controls the DMA's MM2S and S2MM channels: it reads the physical address of each page of a userspace source memory from a scatter/gather list and initiates the DMA MM2S channel to fetch the data of a page each time until all pages are received. Identically, for the outcome data, it reads the physical address of each page of a userspace destination memory from a scatter/gather list and initiates the DMA S2MM channel to send the processed data page by page. It, also, handles DMA MM2S/S2MM interrupts on completion of each transferred page. Two AXI BARs in the PCIe Bridge are dedicated to facilitate concurrent flow of incoming and outgoing data.

In detail, the sequence of operations is as follows. The userspace application allocates two memory areas for storing the initial image data and the processed image data respectively. Then, the application requests from the driver through an ioctl system call to create two scatter/gather lists with the physical addresses of all pages that constitute the two allocated memories in the userspace. The driver directly transfers the two scatter/gather lists over the PCIe to an AXI BRAM inside the FPGA system. Then, the driver activates the Acceleration

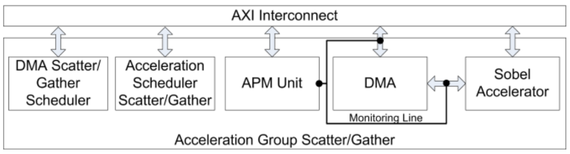

AXI Interconnect
DMA Scatter/ Gather Scheduler
Acceleration Scheduler Scatter/Gather
APM Unit
DMA
Sobel Accelerator
Monitoring Line
Acceleration Group Scatter/Gather

Fig. 5. The Acceleration Group Scatter/Gather (AGSG) block-level organization.

Scheduler Scatter Gather core to initiate the acceleration procedure; this means that this core starts the Sobel Accelerator and then starts the DMA Scatter/Gather Scheduler.

The DMA Scatter/Gather Scheduler reads the physical address of each source page from the AXI BRAM and configures the AXI BAR4 translation register of the PCIe Bridge with that base address; after, it starts the DMA MM2S channel to read data directly from the current page of the userspace source memory. Concurrently, the DMA Scatter/Gather Scheduler reads the physical address of each destination page from the AXI BRAM, sets the AXI BAR5 translation register of the PCIe Bridge with that address and starts the DMA S2MM channel to write processed data directly to the current page of the userspace destination memory.

The DMA performs read burst transactions from its MM2S channel over PCIe directly from the source userspace memory where the image data are located and streams the data though the Sobel Accelerator page by page. The Acceleration Scheduler Scatter/Gather core receives an interrupt from the DMA Scatter/Gather Scheduler on completion of the acceleration and informs the Interrupt Manager to send an MSI interrupt through PCIe to inform the kernel driver for the event, which in turn sends a signal to inform the userspace application that the acceleration procedure is completed.

4 ACCELERATOR ACCESS POLICIES

In order to submit jobs to the accelerators, user-space programs must interface a kernel driver, which ultimately accesses the image filter accelerator groups in the FPGA. In general, by means of extending a typical programming model, the kernel-space driver copies the image data in contiguous kernel-space memory, if the data size adheres to the limitation of four MBytes (else, the AGSG is automatically selected), and then, by using PCIe-DMA transactions, the driver starts the hardware Sobel accelerator. We developed two policies inside the kernel driver, namely the Greedy$_{Single}$ and the Greedy$_{Partition}$, which affect the way that the acceleration groups are distributed and shared in a multi-threaded environment. However, since the main focus of this paper is not the scheduling of hardware resources, we refrain from extended system-level management of the different accelerating methods. Hardware schedulers have been employed to guarantee bounded response times [16] and to manage variable-size workloads [18] or, software orchestrators to provide adaptive resource usage in view of changing application requirements [6].

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. As shown in Figure 6, the driver checks to find the first available acceleration group in the following order:
AGD0→AGD1→AGI0→AGI1→AGI2→AGI3→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.

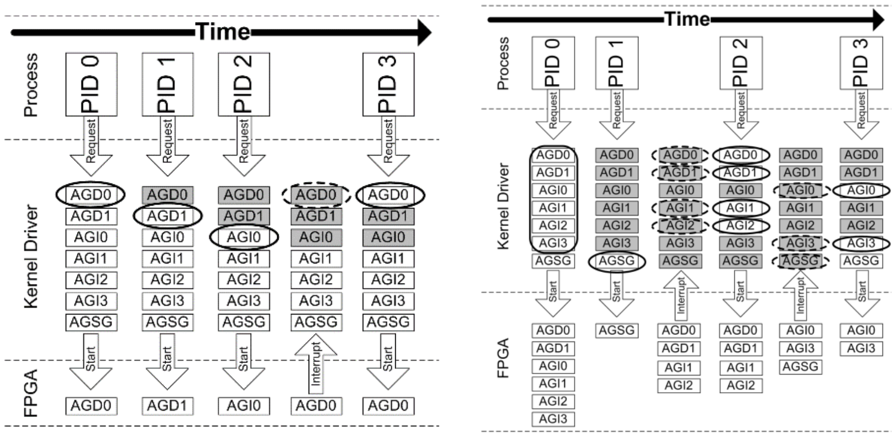


Fig. 6. Allocation of Acceleration Group to user requests in the Greedy$_{Single}$ strategy (left) and in the Greedy$_{Partition}$ strategy (right).

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.

5 ACCELERATOR SYSTEM EVALUATION

By integrating mechanisms to collect performance measurements at runtime we provide metrics that allow not only precise evaluation of the accelerators, but also help to further develop prediction algorithms, self-adaptive and/or orchestrating techniques. To efficiently monitor the acceleration system requires gathering metrics in both the software and hardware layers, namely the userspace application, the kernel driver and the FPGA acceleration units.

In the hardware level, each acceleration group is equipped with an AXI Performance Monitor unit (APM), which uses special hooks to sniff AXI transactions over the DMA AXI master (MM2S/Read) interface, the Sobel AXI Stream output interface and the DMA AXI master (S2MM/Write) interface. In addition, to support timing analysis, an extra APM (*Shared APM*) is introduced as part of the FPGA to serve as a global timer, which is exposed through the PCIe BARs to the kernel driver and the userspace. Thus, not only is it used by the hardware system but also shared by the software components. The acceleration groups occupy special hardware schedulers, which also collect transfer measurements from their dedicated APM and time snapshots from the Shared APM regarding FPGA events.

The userspace applications and the driver use the Shared APM to take time snapshots during the acceleration procedure. The PCIe BARs of the PCIe bridge make the AXI address space of the Shared APM available to the host system. An application creates multiple threads to accelerate image data. Initially, each thread requests from the driver through ioctl calls to allocate shared memory in kernel space that will be used during the execution of the thread to store the measurements that refer to its own acceleration process. Each thread captures the following measurements:

(1) The preparation time which refers to allocating the necessary kernel space memory to store the pre- and post-processed image data.
(2) A starting time snapshot to calculate the total duration of the acceleration at the end.
(3) The latency cost to load the image from the storage device to the kernel space memory.

The driver also captures the time duration that the thread remained in sleep state if no acceleration group was found available.

The Acceleration Schedulers of every active acceleration group collect the following measurements.

- Transferred read/write bytes and total read/write transactions.
- Time duration for CDMA Fetch/Send transfers for any occupied AGI.

These measurements are stored in a specific offset of a local BRAM, independently for each acceleration group. A hardware interrupt signals the completion of an image acceleration and triggers the corresponding driver interrupt routine that is assigned to the acceleration group. The driver interrupt routine searches a singly linked-list to find the measurements memory partition allocated to the process that occupied the acceleration group. Then, it copies the appropriate measurements from the FPGA BRAM to the metrics collection of the corresponding process. In addition, it takes a time snapshot to signal the endpoint of the acceleration itself.

Figure 7 outlines the dynamic monitoring methodology. On completion of the acceleration, the process captures the time duration required for saving the processed image and takes a last time snapshot in order to calculate the total time of the whole procedure. Finally, all the measurements are stored from the kernel memory to an .xml file, which can later be used for data analysis and time visualization.

5.1 System Analysis

In order to analyze the system behavior in depth, to breakdown the overall offloading latency in sub-components and tune each partial operation, we developed a special timing analysis tool which uses .xml files to export statistics and charts, and to subsequently visualize the events that take place during the acceleration procedure.

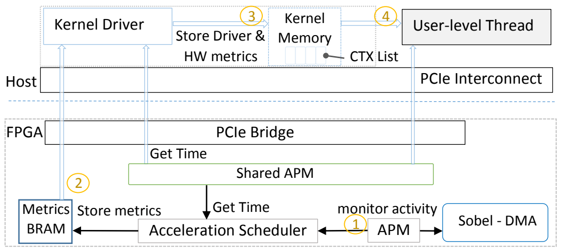


Fig. 7. Monitoring methodology of acceleration events during runtime; numbers depict the process sequence to collect the metrics and store them in the context (CTX) list entry per user thread. Side-band transfers of metrics to the host side has no impact to accelerators' resources.

Table 5. Accelerator system metrics

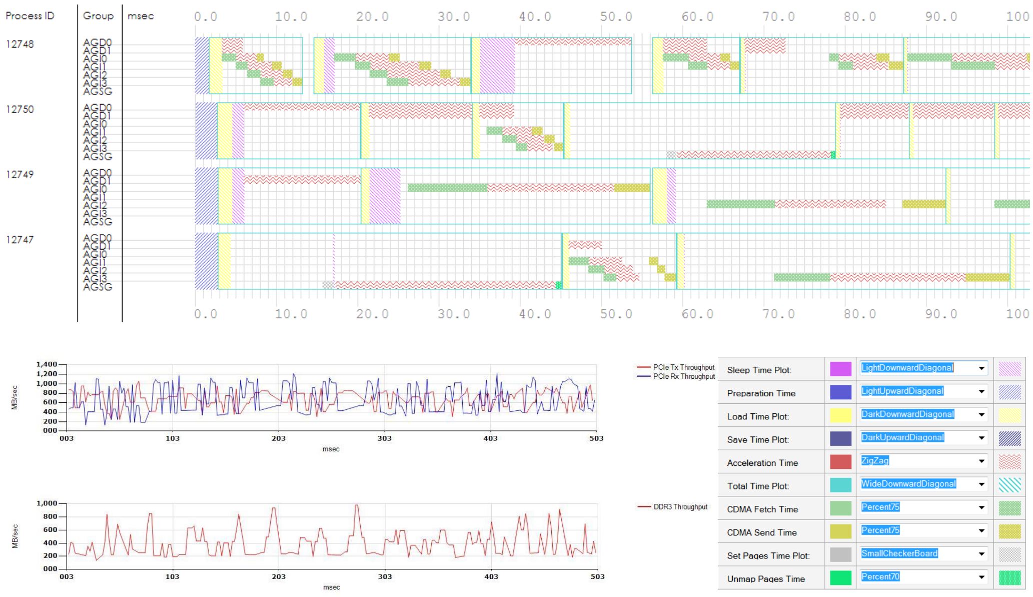
Metric	HW monitor	Handler component	Pattern
Total Duration	Shared Timer	User application	WideDownwardDiagonal
Preparation Duration	Shared Timer	User application	LightUpwardDiagonal
Load Duration	Shared Timer	User application	DarkDownwardDiagonal
Save Duration	Shared Timer	User application	DarkUpwardDiagonal
Sleep Duration	Shared Timer	Kernel driver	LightDownwardDiagonal
Set SG Lists Duration	Shared Timer	Kernel driver	SmallCheckerBoard
Unmap SG Lists Duration	Shared Timer	Kernel driver	Percent70
CDMA Fetch Duration	Shared Timer	HW fetch scheduler	Percent75
CDMA Send Duration	Shared Timer	HW fetch scheduler	Percent75
Accelerator Duration	Shared Timer	Kernel driver	ZigZag
Read Transactions	APM	HW accelerator scheduler	
Read Bytes	APM	HW accelerator scheduler	
Stream Packets	APM	HW accelerator scheduler	
Stream Bytes	APM	HW accelerator scheduler	
Write Transactions	APM	HW accelerator scheduler	
Write Bytes	APM	HW accelerator scheduler	

The analysis tool classifies the events/metrics according to the thread that they correspond. The visualization is based on the events, depicted as rectangles in a timeline according to the duration of each event. There is a dedicated timeline for each thread in the .xml file.

Table 5 summarizes the metrics that are captured by the integrated hardware monitors. By gathering and combining the size of transferred data, the duration of the events and the exact timing we can extract statistics to analyze and optimize:

- Acceleration group utilization
- Acceleration group throughput
- Minimum, average and peak latency per job
- PCIe throughput for each Tx and Rx channel
- Accelerator local memory (DDR3) throughput
- Acceleration group concurrency

Figure 8 shows a snapshot of the major analysis windows that visualizes the breakdown of latencies for four threads which continuously offload jobs (Sobel filtering of 1280x720-sized images) to the accelerators using the Greedy$_{Partition}$ strategy. Sleep time refers to he interval that one thread is put to sleep and in the tail position of a pending queue in the kernel driver when all accelerators are occupied. Notice that in the case that the driver assigns one of the AGI accelerators to a thread that is removed from the queue and is ready to offload the job, there appears a blank interval after the sleep interval in some situations (e.g., threads 12749, 12747); this is due to the CDMA that is shared among AGIs and already used by another thread. The PCIe throughput of each Tx and Rx channel is the instant cumulative throughput of the direct, scatter-gather acceleration groups and of the CDMA engine that is needed by the indirect groups. Figure 8 also depicts the instant DDR3 throughput which is the cumulative throughput due to the AGI0-AGI3 accelerators.



Sleep Time Plot:	🟪	LightDownwardDiagonal	▨
Preparation Time	🟦	LightUpwardDiagonal	▨
Load Time Plot:	🟨	DarkDownwardDiagonal	▨
Save Time Plot:	🟦	DarkUpwardDiagonal	▨
Acceleration Time	🟥	ZigZag	▨
Total Time Plot:	🟦	WideDownwardDiagonal	▨
CDMA Fetch Time	🟩	Percent75	▨
CDMA Send Time	🟨	Percent75	▨
Set Pages Time Plot:	⬜	SmallCheckerBoard	▨
Unmap Pages Time	🟩	Percent70	▨

Fig. 8. Analysis view of a run of four threads using the Greedy$_{Partition}$ policy. The plot is truncated to the first 100 ms for clarity; currently, metrics are used for debugging and performance evaluation purposes, but driver can also exploit metrics to differentiate service at runtime.

6 SYSTEM PERFORMANCE AND HARDWARE COSTS

Due to FGPA space and PCIe AXI BARs constraints, we integrated two AGD, four AGI and a single AGSG accelerators as Figure 9 shows. Each type of accelerator (AGD, AGI and AGSG) uses one pair of AXI BARs, to utilize the six available AXI BARs of the PCIe bridge. The throughput required by a single Sobel processing engine core is almost 343 MB/sec and thus we utilized a PCIe Gen2x4 mode connection that can provide 2 GB/sec throughput with the 8b/10b coding and using a maximum payload size (MPS) of 256B; this configuration delivers a theoretical throughput of 1855MB/s (74.2%) [12]. The PCIe bridge throughput is matched with the total DMA throughput of the system, which is split to 350MB/sec for each AGD accelerator, 250 MB/sec for the AGSG accelerator and almost to 1 GB/sec for the AGI CDMA.

The initially synthesized Sobel core (via using the Vivado HLS tools) is only 4.6% of the size of an accelerator group (695LUTs out of 19160), but delivers only 47MB/s throuhput. Instead of integrating multiple low-throughput acceleration groups, we opted for an optimized

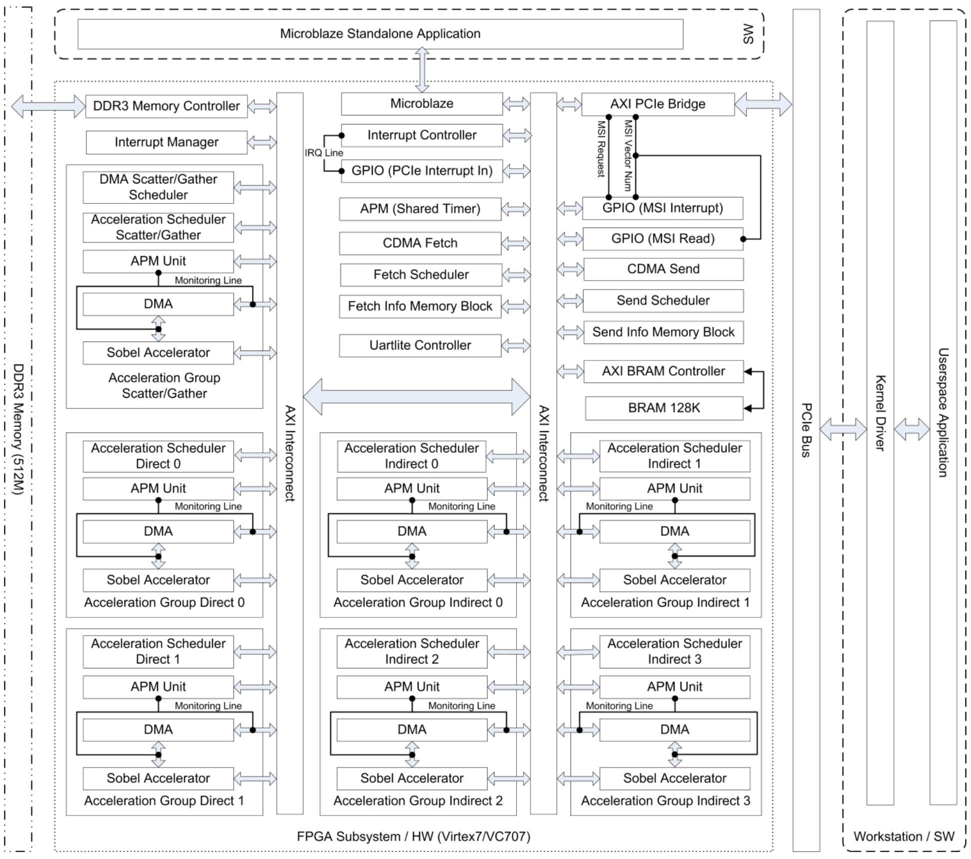

Microblaze Standalone Application
SW
DDR3 Memory Controller
Interrupt Manager
DMA Scatter/Gather Scheduler
Acceleration Scheduler Scatter/Gather
APM Unit
Monitoring Line
DMA
Sobel Accelerator
Acceleration Group Scatter/Gather
Acceleration Scheduler Direct 0
Acceleration Group Direct 0
Acceleration Scheduler Direct 1
Acceleration Group Direct 1
AXI Interconnect
Microblaze
Interrupt Controller
IRQ Line
GPIO (PCIe Interrupt In)
APM (Shared Timer)
CDMA Fetch
Fetch Scheduler
Fetch Info Memory Block
Uartlite Controller
Acceleration Scheduler Indirect 0
Acceleration Group Indirect 0
Acceleration Scheduler Indirect 2
Acceleration Group Indirect 2
AXI PCIe Bridge
MSI Request
MSI Vector Num
GPIO (MSI Interrupt)
GPIO (MSI Read)
CDMA Send
Send Scheduler
Send Info Memory Block
AXI BRAM Controller
BRAM 128K
Acceleration Scheduler Indirect 1
Acceleration Group Indirect 1
Acceleration Scheduler Indirect 3
Acceleration Group Indirect 3
DDR3 Memory (512M)
FPGA Subsystem / HW (Virtex7/VC707)
PCIe Bus
Kernel Driver
Userspace Application
Workstation / SW

Fig. 9. Organization of the multi-threaded, multi-objective accelerator system over PCIe using two AGD, four AGI and one AGSG accelerator groups.

single core design that features 16 parallel Sobel operators (also called as kernel convolvers), resulting in a 343 MB/s processing capacity and with 43% of the area cost of an accelerator group. Initially, the input streaming interface fetches three lines of data before the operators start processing the partition of data that corresponds to each of them. To avoid internal copies of data when the DMA fetches a new row of pixels, we designed virtual indexes to the three rows of data that are active during processing; after the current bunch of pixel rows completes, the new rows are immediately re-named to allow the operators to continue working on the new data. This technique delivered a significant improvement in performance, while plain tiling of parallel Sobel operators only gives little (sub-linear) bandwidth improvement. Finally, the system includes a MicroBlaze CPU, responsible to initialize all components and to reset the shared timer when commanded by the kernel driver. The driver can send an interrupt to the FPGA monitoring system to re-initialize the global accelerators platform time.

6.1 Individual Accelerator Evaluation

In this section, we start our evaluation with the goal to analyze how the latency of the kernel system calls and the communication of the driver with each individual accelerator

Table 6. Average performance metrics (latency in usec) for each single accelerator group for different image sizes (four bytes per pixel). Notice that total latency refers to the actual total delay that the application experiences.

	32×32 (4,096 B)	QVGA (307,200 B)	VGA (1,228,800 B)	HD (3,686,400 B)
AGD Acc Latency	41.76	988.54	3584.62	10221.6
Total Latency	89.42	1086.73	3761.15	10697.8
AGI Acc Latency	40.41	988.09	3584.26	10218.5
CDMA Fetch	8.68	341.42	1351.7	4072.5
CDMA Send	7.14	310.24	1231.85	3689.4
Total Latency	128.62	1741.4	6355.88	18479.9
AGSG Acc Latency	45.50	1286.15	4890.40	13439.9
Set Pages	3.1	82.22	381.77	1369.6
Unmap Pages	3.30	76.77	234.13	711.2
Total Latency	551.80	1971.73	6420.98	17142.6

impacts the computation capacity of the accelerator from the perspective of the application that waits for the processed data. In this direction, we capture metrics by activating only a single accelerator at a time, and by using the kernel driver to send data sets that scale in size from a tiny image of 32×32 pixels up to an HD image. By introducing a tiny image (4K bytes) we observe the result of the above overheads that dominate the accelerator latency. As table 6 summarizes the results of this evaluation strategy for all methods, the benefit of the accelerator in terms of performance is clear as the amount of data grows. The total latencies shown have a standard deviation that is almost the same for the AGD and AGI accelerators, 9.8 and 12.3 respectively and a deviation that scales from 14.6 to 250.8 for the AGSG case; AGSG is more sensitive to OS.

The total latency can be roughly decomposed to the accelerator latency, to the data transfers and to the interaction with the operating system (kernel driver calls, interrupt handling etc). Figure 10 abstracts the details of each sub-process and data flow for each accelerator type. Of course, the role of the accelerator controller differs across each type. The AGD and AGSG are located in the shaded box, while an ADI includes also the CDMA and accesses the external DDR3 memory.

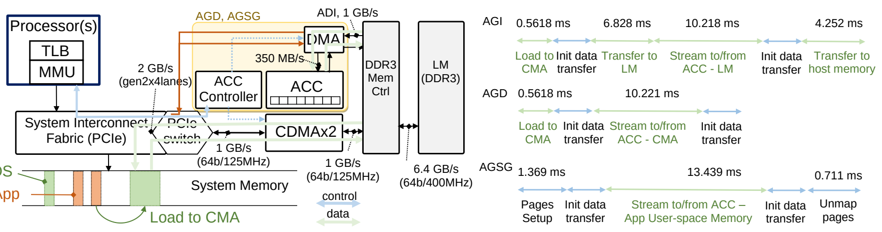


Fig. 10. Block-level organization of accelerator system connected to PCIe and an example breakdown of latency for processing an HD image using AGD, AGI and AGSG methods.

Figure 11 on the left depicts the impact of the operating system, which is significant for small amounts of data, and in the case of the AGSG group, due to the virtual to physical translation of the user pages for the source and destination data.

To capture the operational intensity of each accelerator method against the memory traffic and against the communication activity, we collected the actual latency of each computation engine, i.e., of each Sobel core, when it is activated, and the total latency of each offload process to complete. Hence, similar to the roofline model, commonly used to depict the peak platform performance and potential bottlenecks [33][22], we plot the gathered metrics for the various workloads in the roofline graph on the right. As shown in Figure 11 on the right, the AGD group exhibits the best ratio as the workload scales. On the contrary, the AGI accelerator experiences low ratio with increasing data because of the unavoidable transfers from and to the local DDR3; this latency also scales up with the data size. Hence, in comparison to the ADI group, the AGSG group, although it requires more 'effort' by the OS initially, the direct data transfers from the user pages compensates the overhead of virtual to physical address space translation. Finally, it is important to note that applications with a large gap between the accelerator computation throughput (i.e., operational intensity) and data-transfer bandwidth and OS latency suffer more performance impact from data transfer and kernel driver latencies.

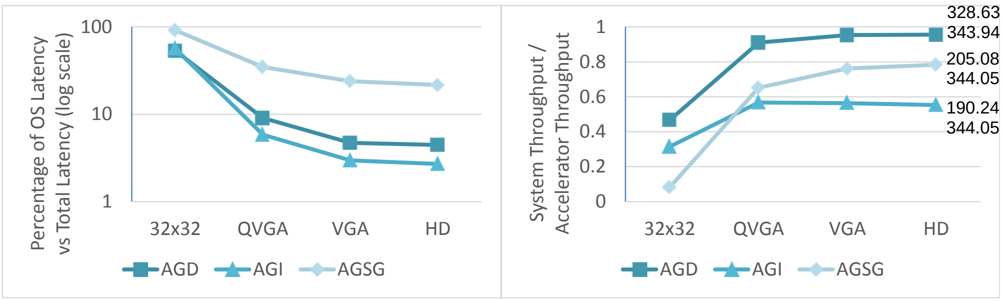


Fig. 11. Percentage of Operating System latency over total latency (in log scale) for each individual accelerator group (left plot). Computation throughput vs total throughput for each accelerator group (right plot); throughput is shown in MB/s only for the HD case.

6.2 Full Accelerator System Evaluation

Figure 12 shows the comparative performance when executing Sobel acceleration on HD images of 1280x720 (720p) pixels. We used the Greedy$_{Single}$ and Greedy$_{Partition}$ policies that the driver enforces to handle job dispatching of multi-threaded applications.

After extensive verification through thousands of tests (each test case tested by 100 times for a total of 12,000 runs), Figure 12 summarizes the average latency of a single image in the left plot, and the accelerators system throughput in the right plot when the workload scales from 1 to 32 images. For reference, a single threaded Sobel application has a latency of 64 ms to complete on the host, an Intel Xeon E3-1246 v3, 8MB cache at 3.5 GHz, with 16GB DDR3 clocked at 1866MHz and runs Debian (Jessie), 3.16.0-4-amd64 kernel. Notice that the Greedy$_{Partition}$ scenario is an optimum solution when we have concurrent acceleration requests by one or two threads, while the rest cases exhibit minor performance differences in both policies.

For the two dispatching policies, Figure 13 shows the PCIe average and peak throughput in the left plot and the accelerators DDR3 average and peak throughput in the right plot. Notice that for the Greedy Single policy the DDR3 is never used in the case of one or two threads since only the AGD accelerators are activated. In all cases the average measurements

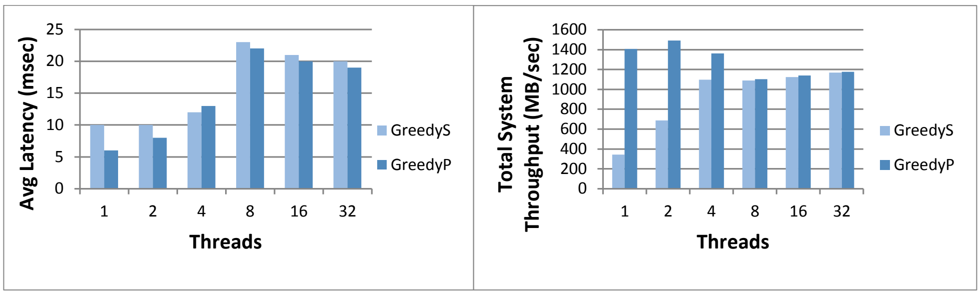


Fig. 12. Average latency for a single image and total system throughput when applying the Greedy$_{Single}$ (GreedyS) and Greedy$_{Partition}$ (GreedyP) dispatching policies on HD images (1280x720) and when scaling the number of threads.

are much smaller compared to the peak values due to the size of the images, which size leaves small possibilities to find many accelerators idle and thus simultaneously activate them. Even in the case of the Greedy$_{Partition}$ strategy, that accelerators process portions of the same image, peak throughput of PCIe and of DDR3 traffic is rarely reached; as shown in table 6, a large fraction of time to process an image is consumed by the CDMA fetch and send operations. Hence, even though we used two such copy engines, still, the four accelerators of the AGI group are not fully utilized.

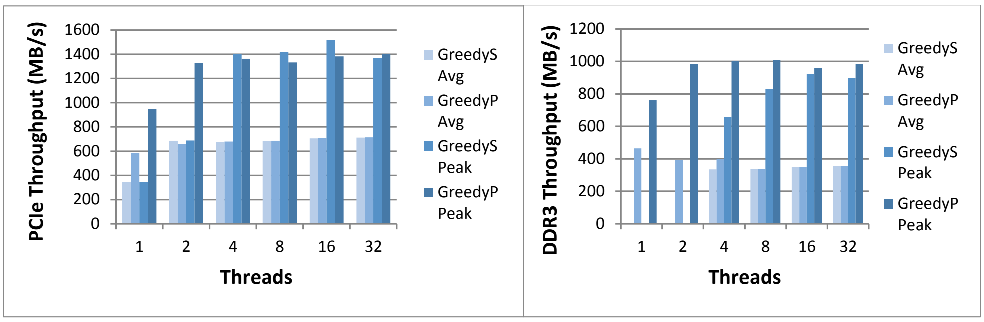


Fig. 13. Average and Peak throughput of the PCIe bridge and of the FPGA DDR3 memory for the Greedy$_{Single}$ (GreedyS) and Greedy$_{Partition}$ (GreedyP) policy when scaling the number of threads.

Figure 14 shows the system throughput in terms of Jobs/sec for both policies. The leftmost plot refers to workloads of only HD images. The Greedy$_{Single}$ strategy delivers better scaling when the threads that utilize the accelerators are less than the total number of accelerators. It is notable though that, for significantly larger thread numbers, both policies deliver similar throughput. The plot in the middle and on the right show how the system throughput scales for workloads of different size, that is, for VGA images (640×480) and QVGA images (320×240). Metric Jobs/sec is normalized to that of a single thread. As expected the Greedy$_{Single}$ policy scales better as the data size becomes smaller.

Table 7 summarizes the area cost of each accelerator group and of the full FPGA system. All special-purpose components (schedulers and accelerators) are developed using Vivado High-Level-Synthesis tools. Each acceleration group consumes equivalent resources, while the scheduling and control components overwhelm the cost of the Sobel accelerator itself. The accelerator system operates at 125 MHz using 128-bit wide data-paths internally. Despite

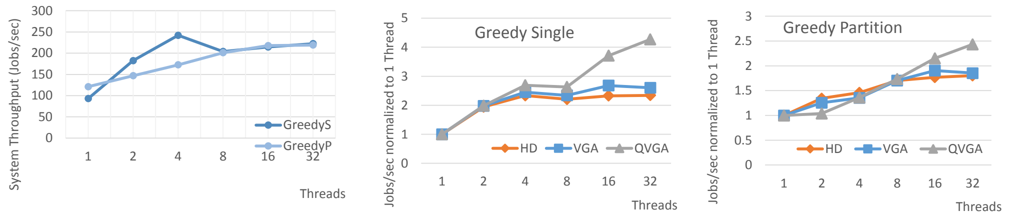


Fig. 14. Scaling of jobs/sec for the Greedy$_{Single}$ (GreedyS) and Greedy$_{Partition}$ (GreedyP) policies when increasing the number of threads (left, using only HD images). Scaling for different workload sizes for the Greedy$_{Single}$ and the Greedy$_{Partition}$ policies (middle and right).

the modest clock frequency, the aggregate processing capacity of the accelerators is matched to the supported throughput of the PCIe Gen2 x4 lanes. After implementing the design on the VC707 platform, we collected power figures using the Vivado reporting power; on the basis of a typical process and 50% switching activity the total on-chip power is 5.063 W. To estimate the usage of only AGD-type accelerators, we imported this configuration in the Xilinx XPE power estimator and by disabling the IO pins for the DDR3, the power is reduced almost by 20% (4.037 W).

Using the power figures and the breakdown of resources as depicted in table 7 we calculate the power of each accelerator including the required circuitry (dynamic), the transceivers power, the I/O and the chip leakage, proportionally to the percentage of area. Then, from the actual host, which is enabled with the accelerators platform, we collected the utilization of each accelerator group and the duration that is active with a precision of ns, via the hardware monitors and via our analysis-visualization tools. The intensity of the activity of the accelerators when iterating for twenty jobs for all active threads is normalized to the workload of a single thread that utilizes only one specific accelerator group, i.e., AGD, AGI or AGSG. Therefore, we extracted the energy required by each accelerator group when scaling the workload from 20 jobs for a single thread up to total of 640 jobs dispatched by 32 threads. Figure 15 shows the summary of the collected measurements for the utilization percentage and the corresponding consumed energy inside the FPGA device. It is important to note that the energy consumed by the external DDR3 in the AGI case is not depicted in the plot. Due to the applied policies, the Greedy$_{Single}$ is more efficient for small workloads as it utilizes more the AGD groups, while for large workloads both policies have similar effects, with the accelerator sub-system energy to be consumed mostly by the AGI accelerators, even though the AGD usage dominates the distribution of jobs due to the adopted offload strategies.

Consequently, the proposed design variants allow the system to dynamically adjust the performance and energy consumption of the data center on the basis of runtime energy constraints, via deactivating the AGIs for instance, or maximize the opportunities of data reuse or accelerator components scaling (depending of the application of course) through exploiting local memory. The offloading policies inside the kernel can easily be adapted either to consider system energy constraints, and/or accept user-space hints for promoting user service priority. We intend to explore such possibilities in future work.

7 SUMMARY, CONCLUSIONS AND FUTURE WORK

Design methods that enable hardware accelerators to provide energy efficiency through access to user data with zero copying and use of on-chip scratchpad memories, and to provide

Table 7. Hardware cost of accelerator groups using a Virtex-7 XC7VX485T FPGA

Name	Slice LUTs	LUTs (Memory)	LUTs (Logic)	BRAMs	DSPs
Accel. System (2 AGD, 4 AGI, 1 AGSG)	256205	12507	243698	641.5	24
Utilization (%)	84.3	9.5	80.3	62.3	0.8
Acceleration Group Direct	19221	571	18650	61.5	3
Acceleration Scheduler Direct	3317	147	3170	0	3
AXI PMU[a]	4508	288	4320	0.5	0
DMA	1984	136	1848	5	0
AGD AXI Interconnect	158	0	158	0	0
Sobel Filter Accelerator	8689	0	8689	56	0
Acceleration Group Indirect	20137	578	19827	61.5	3
Acceleration Scheduler Indirect	4372	154	4218	0	3
AXI PMU	4795	288	4507	0.5	0
DMA	2124	136	1988	5	0
AGI AXI Interconnect	158	0	158	0	0
Sobel Filter Accelerator	8690	0	8690	56	0
Acceleration Group Scatter/Gather	21491	664	20827	61.5	6
Acceleration Scheduler Scatter/Gather	2640	118	2532	0	3
AXI PMU	4506	288	4228	0.5	0
DMA	1986	136	1850	5	0
DMA Scatter/Gather Scheduler	2835	122	2713	0	3
AGSG AXI Interconnect	179	0	179	0	0
Sobel Filter Accelerator	8696	0	8696	56	0
Fetch Scheduler	2956	130	2826	0	0
Send Scheduler	3072	131	2941	0	0
Fetch/Send Memory Info Block	1009	0	1009	0	0
CDMA fetch	1269	89	1180	0	0
CDMA send	1273	89	1184	0	0
Shared APM	1219	130	1089	0	0
PCIe bridge	18253	307	17946	0	0
DDR3 controller	14064	2325	11739	0	0
AXI Interconnect PCIe mig	5938	1254	4684	0	0
AXI Interconnect main	58747	2969	55778	0	0
AXI Interconnect DMAs	4681	719	3962	0	0
Microblaze	1172	121	1051	0	0

[a]AXI Performance Monitor Unit (APM)

scalability for extending performance, are indispensable to address diverse requirements for a multitude of workloads. Today, thanks to high-level synthesis tools, accelerator-centric design is easier and more widely-used than ever before. By matching communication methods over PCIe to workload characteristics and system constraints and, by exploiting scheduling and parallelized, pipelined DMA transfers, all controlled by hardware customized components at the FPGA fabric side, we demonstrated a high-performance FPGA accelerating system suitable for efficient acceleration of computation intensive tasks of streaming applications.

Today, common hardware accelerators, as expected, accelerate computation greatly while leaving communication, control and programmability mostly intact. We demonstrated that tackling CPU-Accelerators overheads, such as user- kernel- copies, costly IOMMUs, and

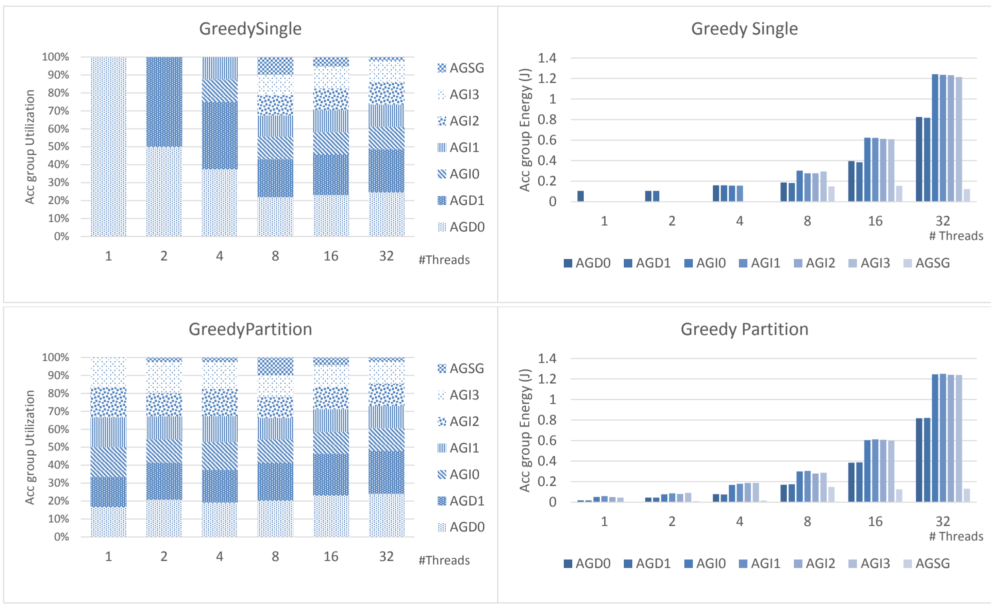


Fig. 15. Breakdown of accelerator system energy for Greedy$_{Single}$ and Greedy$_{Partition}$ policies when scaling the number of threads. For the utilization of the Accelerator groups as shown on the left plots, total device (XC7VX485T) energy is plotted on the right graphs; notice that the DDR3 energy required by the AGI groups is not included.

interrupt management, and dynamically adapting accelerator resources to the needs of datacenter applications are key important factors that accelerator designers can successfully address. At the same time, the CPU is relieved from the responsibility for orchestrating the entire system, including synchronization of accelerators and DMAs for coordinated data access and data processing. Overall, the proposed framework that is open-sourced in *http://isca.teicrete.gr/self-adaptive-hsa/* is the first to our knowledge to easy the burden of both integrating multiple accelerators (based on standard Xilinx AXI PCIe bridge cores), freeing from intricacies of PCIe protocol, and the burden of selecting the optimal communication method at the programmer side, given the variable needs of datacenter workloads. While we selected image filtering as the baseline application domain in this paper for its streaming data communication model between the processor(s) and the accelerators, the key insights of this work are applicable to other domains with modest adjustments.

Our future plans involve exploration with different architectural organizations of the accelerator groups (e.g. two AGSGs and AGDs) and the development of additional policies, mostly in concert with operating system scheduling and resource management policies for energy efficient computing in a datacenter environment.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their time and valuable feedback. The work is partially supported by the EU FP7 SAVE project under Grant No.: 610996 and National Contribution Support Program 2014-2016 SAVE No.80278.

REFERENCES

[1] Michael Hollinger Brad Brech, Juan Rubio. 2014. Data Engine for NoSQL - IBM Power Systems Edition. White Paper. (2014). https://www-304.ibm.com/webapp/set2/sas/f/capi/CAPI_FlashWhitePaper.pdf

[2] Tony M. Brewer. 2010. Instruction Set Innovations for the Convey HC-1 Computer. *IEEE Micro* 30, 2 (March 2010), 70–79. https://doi.org/10.1109/MM.2010.36

[3] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 109, 6 pages. https://doi.org/10.1145/2897937.2897972

[4] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2015. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Proceedings of the 52Nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, Article 202, 6 pages. https://doi.org/10.1145/2744769.2744794

[5] David de la Chevallerie, Jens Korinth, and Andreas Koch. 2016. ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators. *SIGARCH Comput. Archit. News* 43, 4 (April 2016), 34–39. https://doi.org/10.1145/2927964.2927971

[6] G. Durelli, M. Coppola, K. Djafarian, G. Kornaros, A. Miele, M. Paolino, Oliver Pell, Christian Plessl, M. D. Santambrogio, and C. Bolchini. 2014. *SAVE: Towards Efficient Resource Management in Heterogeneous System Architectures*. Springer International Publishing, Cham, 337–344. https://doi.org/10.1007/978-3-319-05960-0_38

[7] Jian Gong, Jiahua Chen, Haoyang Wu, Fan Ye, Songwu Lu, Jason Cong, and Tao Wang. 2014. EPEE: An Efficient PCIe Communication Library with Easy-host-integration Property for FPGA Accelerators (Abstract Only). In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA '14)*. 255–255.

[8] Yuchen Hao, Zenhman Fang, Glenn Reinman, and Jason Cong. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 37–48. https://doi.org/10.1109/HPCA.2017.19

[9] Joel Hestness, Stephen W. Keckler, and David A. Wood. 2015. GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC '15)*. IEEE Computer Society, Washington, DC, USA, 87–97. https://doi.org/10.1109/IISWC.2015.15

[10] Intel. 2009. An Introduction to the Intel QuickPath Interconnect. White Paper. (Jan 2009). www.intel.com/technology/quickpath/introduction.pdf

[11] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators. *ACM Trans. Reconfigurable Technol. Syst.* 8, 4, Article 22 (Sept. 2015), 23 pages.

[12] Jason Lawley. 2014. Understanding Performance of PCI Express Systems. (2014). Xilinx, WP350 (v1.2) Oct 28.

[13] Qiwei Jin, Diwei Dong, Anson H. T. Tse, Gary C. T. Chow, David B. Thomas, Wayne Luk, and Stephen Weston. 2012. Multi-level Customisation Framework for Curve Based Monte Carlo Financial Simulations. In *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC'12)*. Springer-Verlag, Berlin, Heidelberg, 187–201. https://doi.org/10.1007/978-3-642-28365-9_16

[14] Christoforos Kachris and Dimitrios Soudris. 2016. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–10. https://doi.org/10.1109/FPL.2016.7577381

[15] A. Kegel, P. Blinzer, A. Basu, and M. Chan. 2016. IOMMU: Virtualizing IO through IO Memory Management Unit (IOMMU). In *Tutorial in ASPLOS-XXI*.

[16] George Kornaros. 2013. High-Speed Hardware Arbitration Supporting Priorities and Bounded Service Latency. *IEEE Embedded Systems Letters* 5, 2 (June 2013), 21–24. https://doi.org/10.1109/LES.2013.2251454

[17] George Kornaros, Kostantinos Harteros, Ioannis Christoforakis, and Maria Astrinaki. 2014. I/O virtualization utilizing an efficient hardware system-level Memory Management Unit. In *2014 International Symposium on System-on-Chip (SoC)*. 1–4. https://doi.org/10.1109/ISSOC.2014.6972448

[18] George Kornaros and Menelaos Pratikakis. 2016. VWQS: A dispatching mechanism of variable-size tasks in heterogeneous systems. In *2016 International Conference on High Performance Computing Simulation (HPCS)*. 196–203. https://doi.org/10.1109/HPCSim.2016.7568335

[19] Alex Markuze, Adam Morrison, and Dan Tsafrir. 2016. True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 249–262. https://doi.org/10.1145/2872362.2872379

[20] Michal Nazarewicz. 2012. A deep dive into CMA. (2012). https://lwn:net/Articles/486301/ LWN article.

[21] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/3020078.3021740

[22] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Puschel. 2014. Applying the roofline model. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 76–85. https://doi.org/10.1109/ISPASS.2014.6844463

[23] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-defined accelerator for largescale dnn systems. In *2014 Hot Chips 26 Symposium (HCS)*. 1–23. https://doi.org/10.1109/HOTCHIPS.2014.7478821

[24] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[25] Sophia Shao, Yakun, Likun Xi, Sam, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783751

[26] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* 59, 1 (Jan. 2015), 7:1–7:7. https://doi.org/10.1147/JRD.2014.2380198

[27] Nasibeh Teimouri, Hamed Tabkhi, and Gunar Schirner. 2015. Revisiting Accelerator-rich CMPs: Challenges and Solutions. In *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, Article 84, 6 pages. https://doi.org/10.1145/2744769.2744902

[28] Ben van Werkhoven, Jason Maassen, Frank J. Seinstra, and Henri E. Bal. 2014. Performance Models for CPU-GPU Data Transfers. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 11–20.

[29] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171.

[30] Malte Vesper, Dirk Koch, Kizheppatt Vipin, and Suhaib A. Fahmy. 2016. JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–9. https://doi.org/10.1109/FPL.2016.7577334

[31] Markus Weinhardt, Bernhard Lang, Frank M. Thiesing, Alexander Krieger, and Thomas Kinder. 2015. SAccO. *Microprocess. Microsyst.* 39, 7 (Oct. 2015), 543–552.

[32] Xilinx. 2017. AXI Memory Mapped to PCI Express (PCIe) Gen2 v2.8 Logicore IP Product Guide (PG055). (2017).

[33] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

Received September 2017