Enabling Efficient Job Dispatching in Accelerator-extended Heterogeneous Systems with Unified Address Space

Georgios Kornaros* and Marcello Coppola†
*Department of Informatics Engineering, Technological Educational Institute of Crete, Heraklion 71500, Crete, Greece
Email: kornaros@ie.teicrete.gr
†STMicroelectronics, Grenoble, France

***Abstract*—In addition to GPUs that see increasingly widespread use for general-purpose computing, special-purpose accelerators are widely used for their high efficiency and low power consumption, attached to general-purpose CPUs, thus forming Heterogeneous System Architectures (HSAs). This paper presents a new communication model for heterogeneous computing, that utilizes a unified memory space for CPUs and accelerators and removes the requirement for virtual-to-physical address translation through an I/O Memory Management Unit (IOMMU), thus making stronger the adoption of Heterogeneous System Architectures in SoCs that do not include an IOMMU but still representing a large number in real products. By exploiting user-level queuing, workload dispatching to specialized hardware accelerators allows the removal of drawbacks present when copying objects through using the operating system calls. Additionally, dispatching is structured to enable fixed-size packet management that hardware specialized logic accelerates. To also eliminate IOMMU performance loss and IOMMU management complexity, we propose direct accelerator data placement in contiguous space in system-memory, where, the dispatcher provides trasparent access to the accelerators and at the same time offers an easy abstraction in the programming layer for the application. We demonstrate dispatching rates that exceed ten thousand jobs per second implementing architectural support on a low-cost embedded System-on-Chip, bounded only by the computing capacity of the hardware accelerators.**

I. Introduction

In recent years, as well as the trend for embedding multiple cores in a single chip, an architectural trend is also the growing prominence of heterogeneous architectures [1][2]. Today, processor manufacturers including AMD, Intel, and NVIDIA integrate CPUs and GPUs on the same chip while a coalition of companies including AMD, ARM, Qualcomm, and Samsung recently formed the Heterogeneous System Architecture (HSA) Foundation to better support heterogeneous computation. Moreover, system-level accelerators such as Intel Phi processing engines or other FPGA-based coprocessors are emerging for High-Performance Computing (HPC) in the form of heterogeneous platforms to deliver highly parallel processing. To address efficiency in terms of performance/power ratio, the use of on-chip accelerators in multi-, many-core systems is becoming more popular [3]. As scaling the number of processors does not always translate to linear speedup, on-chip application-specific hardware components are increasingly employed to improve the overall system performance; such custom accelerators offer power-efficient implementations of a particular functionality. However, even though physically putting together CPUs and GPUs or other accelerators on the same chip or platform, the available programming models still consider them separated.

This strong need towards heterogeneous computing has driven the adaptation of applications mostly engineering to make effective use of multi-core CPUs and massively parallel GPUs using toolkits such as Threading building blocks (TBB), OpenMP, CUDA, OpenCL [4][5], and others like them. Each offers a different approach to parallelization [6], exploiting for instance an intuitive fork-join model and enabling locality-friendly coarse grained parallelism (OpenMP), or expressing fine-grained parallelism (OpenCL), enabling for example, implicit or explicit vectorization at the compiler level. Additionally, to amortize the overhead of offloading, large kernels are suggested and throughput-oriented code to more easily hide the latency and variability of each individual operation. At the same time, unifying the memory space has emerged in GPGPU computing with features such as shared virtual memory and demand paging [7]. Unfortunately it comes at a price, and that is performance. Automatic memory management is convenient but suffers from many drawbacks, preventing heterogeneous systems from achieving their full potential [8].

The concept to offload a kernel task to throughput-optimized accelerators or CPU/GPU combination is shown in figure 1. Efficient task dispatching and reduction of copying overheads present important challenges in this heterogeneous computing style, especially when the job mix in not known in advance. Accelerators can hardly offer preemptive computing or unified address space which raises the importance of job scheduling and of fast methods for data transfers.

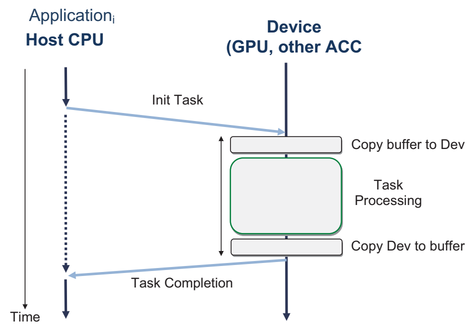

Fig. 1. Flow of execution on heterogeneous CPU-Accelerator environment

In this paper, we describe an infrastructure to facilitate efficient heterogeneous processing in Systems-on-Chip (SoC) with no need for an I/O Memory Management Unit (IOMMU), by simplifying the programming of hardware accelerators and by utilizing efficient data sharing between host CPU and tightly-coupled hardware accelerators. HSA-driven initiative proposes sharing of virtual address space between all processing components in the system in order to remove the need of explicit copies [9]. Similarly, our architecture supports *unified memory addressing*, virtual and/or physical, which allows the host CPU, the accelerators and the dispatching mechanism that we present hereafter to reference the same virtual and physical memory space. Our solution mainly targets SoC architectures and embedded systems that do not include IOMMUs, yet, the integrated accelerator components can serve applications in their own virtual address space. With the decline in memory price the time of scarcity for memory fades away. Hence, our infrastructure exploits part of system memory space to avert using an IOMMU and remove its overheads (maintenance and synchronization with CPU's MMU, latency in multi-level page-table walking and silicon cost).

Finally, we avoid OS driver latency and memory copies by accessing the dispatch subsystem at user-level. We deliver easy programmability while at the same time we address the reasons of latencies in today's communications, kernel launch and data transfer that introduce delays and variabilities, which are added no matter the size of the kernel.

The rest of the paper is organized as follows. Section II discusses background work in optimizing CPU-Accelerator communication and in dispatching methods. Section III introduces the hardware and software design of the proposed framework, while section IV presents the full realization of an accelerator extended embedded system using the ZYNQ System-on-Chip. Section V evaluates the proposed dispatching framework and analyzes the key features in relation to close proposed solutions and finally section VI concludes the paper.

II. Preliminaries and Related Work

Applications in heterogeneous architectures take advantage of hardware accelerators and GPUs, by offloading computations, intensive portions of their execution to the hardware accelerator, programmable or not, or to the GPU. These offloaded computation tasks are referred in this paper as *kernels* in order to distinguish from OS kernel that is central part of an operating system. When the CPU dispatches a task to the hardware accelerator or GPU, it is usually necessary to pass through an OS service and an OS kernel driver before finally reaching the final target, which causes non-negligible performance degradation.

When we offload a computation, the data to be used must be moved from the memory of the CPU to the memory of the accelerator device. Only at this stage, data can be used by the accelerator or by the GPU. This is commonly implemented by a DMA transfer. This operation requires a pointer to data (i.e., a virtual address, *UserVA*) and a size in bytes, as well as one or more destination addresses depending on the size of the buffer to be transferred. Then, the OS kernel translates the userVA to an OS kernel virtual address (*kernelVA*)and next, this kernelVA to a list of physical pages (*PA*) and makes sure they are ready to be transferred by pinning the memory. The OS kernel driver uses the list of physical pages to program the device's DMA engine(s). After the transfer is completed, the runtime communication library should eventually clean up any resources used to pin the memory. After the *kernel* has been executed by the accelerator, the processed data must be moved back to the host memory following the same ping-pong of buffers. This ping-pong of buffers introduces significant performance penalty due to intermediate copies.

To address this challenge, technologies mainly targeting GPU such as GPUDirect RDMA emerge, attempting to enable a direct path to speed up data transfer between the GPU and a third-party peer device using standard features of PCI Express [10]. The mechanics define how a PCIe device can issue reads and writes to a peer device's Base Address Registers (BAR) addresses in the same way that they are issued to system memory, essentially trying to facilitate low overhead exchange of memory mapped I/O addresses. However, RDMA includes some disadvantages due to inconsistent updating of information between CPU and GPU. Without a technique called pinning, elements of memory systems can get corrupted in RDMA-enabled setups. Moreover, mircoservers are developed with NUMA-based architectures that attempt to optimize remote accesses [11]. IBM introduced the CAPI interface to provide a high-performance solution for implementing computation-heavy algorithms on FPGAs, but is specific to POWER8 CPU [12]. At the same time, even optimized IOMMUs (System MMU by ARM and our prior IOMMU design [13]) attempt to provide a unified view of virtual address space to the accelerators; however, this comes at the cost of latency overheads due to costly I/O TLB management and CPU interrupting [14][15].

Several researchers explored GPU scheduling [16][17]. Kato [16] et al., implement a priority-based scheduling policy using ring buffers in kernel space for multi-tasking environment, based on monitoring GPU commands issued from user space. However, for small, frequent acceleration requests, the overhead of trapping to the kernel can make this approach problematic.

In another approach, communication libraries such as GASNet, an existing Partitioned Global Address Space (PGAS) communication API, attempt to provide a unified programming model and API for all components in a heterogeneous system. Such research efforts attempt to bring performance and efficiency to the applications that execute in heterogeneous processing elements, while struggling to reduce latencies of software stack or even by introducing remote memory access hardware components [18]. Hardware support for optimizing different ISA-based heterogeneous systems is also proposed in a variety of contexts, by accelerator management to mitigate memory latencies during data transfer [19], or by using hardware assisted scheduling of variable-sized jobs [20], or by optimizing intranode communication using DMA assistance [21]. By sharing the virtual address space of CPU and accelerators, researchers have recently proposed a user-level library, such as GMAC, to make heterogeneous systems easier to program while reducing performance penalties [1][22][23]. It is not though guaranteed to successfully map the accelerator's memory to the same range of virtual memory address space. To provide programmers the illusion of unified CPU and GPU memory, runtimes are developed that automatically migrate data in and out of the GPU memory [7]. Despite the advancement in programmer convenience, initial investigations on real hardware show that the performance overhead of GPU paged memory may be significant [24].

Recently, proposals involve mixed hardware-software designs which rely on placing the shared data in contiguous kernel-space memory and replace the standard malloc() with a customized implementation [25]. However, the overhead of equiping the accelerators with a customized DMA and TLB is large, while at the same time the kernel-space device driver is complex enough to handle these.

In this work we describe an infrastructure, the first to the best of our knowledge, that utilizes a unified address space among host processors with different ISA and hardware accelerators, which infrastructure in addition combines hardware support with user-level queuing targeting SoCs with no IOMMU, which differentiates significantly from HSA specification and evaluation frameworks [9][15][26]. This proposed architecture delivers a programming model which conveniently helps the programmer to address any hardware accelerator in a uniform manner.

III. Dispatching to Hardware Accelerators

This section presents the key components of the developed system architecture inspired by the HSA programming model in SoCs with no IOMMU. Hardware accelerators, referred by HSA as HSA components, are interfacing the host processor generally referred as HSA agent via a hybrid dispatcher, named hereafter *Generic Packet Processing Unit* (*GPPU*). Essentially, the GPPU consists of both hardware components and a *Runtime* that provides the interfaces necessary for the Host CPU to launch compute kernels to the on-chip accelerators. In a typical software architecture stack for programmable hardware-specific accelerators, which are implemented using configurable processors such as the ones by Tensilica[27], we need to transfer the data and to program the hardware accelerator in order to execute the specific functionality, for instance program an H264 decoder accelerator or perform the I/O operation. Figure 2 shows an abstract view of a system that provides dispatching support via GPPU components. Single or multiple accelerator units, called hereby HSA on-chip components can be even connected to a single GPPU. The Runtime includes userspace and operating system services to enable an efficient offloading mechanism not only to on-chip hardware accelerators but also from an accelerator to another processing component (CPU, GPU, or custom accelerator).

The GPPU supports user-level command queues that are allocated at runtime. Each queue contains packets (commands) as defined in the Architected Queuing Language (AQL packets) and they are allocated and de-allocated by applications

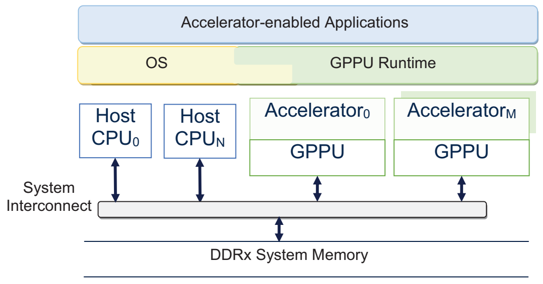

Fig. 2. System organization including GPPU-enabled application-specific processing units as accelerators

through a specific GPPU runtime infrastructure. As defined by HSA [9], queues are semi-opaque objects: there is a visible part that is represented by the queue context and the ring buffer (pointed to by *baseAddress*) and the invisible part, which contains the read and write indexes as illustrated in figure 3. The ring buffer can be directly accessed by the application to initialize the attributes of the job to offload, while the read and write indexes of the queue can be only accessed using the specific GPPU runtime.

The host CPU packs a single or a sequence of commands to fixed size packets. The structure of the packet is known to the applications. The HSA system architecture specification [9] defines three packet formats, kernel dispatch, agent dispatch[1] and barrier, while in this paper we consider only a single *dispatch* packet and the *barrier* packet. Figure 3 shows the generic job offloading process to an accelerator and depicts the dispatch packet format, which is used to launch the kernel via the pointer that is included in the dispatch packet to the on-chip accelerator.

The GPPU performs the following functions: i) it manages requests from user-level applications to offload jobs to accelerator engines; the requests are performed in the form of packets which are placed in circular queues until the GPPU serves these queues ii) it mediates the process to offload kernel jobs and data from user application memory space to a hardware accelerator and deliver the outcomes of these jobs iii) it synchronizes concurrent events and signal events to trigger and notify the HSA agent.

A user-level queue is a shared memory space between the Host CPU (i.e., HSA Agent) and the GPPU that is used to implement one-way communication from the Host CPU to the GPPU. Both, Host CPU and GPPU have to maintain an internal state able to read and write to the command buffer in a consistent way. Intranode communication to achieve a complete offload operation involves the *launch*, *active* and *completion* phases. The communication protocol between an application that exploits user-mode dispatching through the GPPU hardware support and the hardware accelerator is outlined in figure 4.

The Host CPU initializes the queue contexts in the GPPU in order to configure the communication protocol parameters. A

[1]to distinguish the sender of the packet

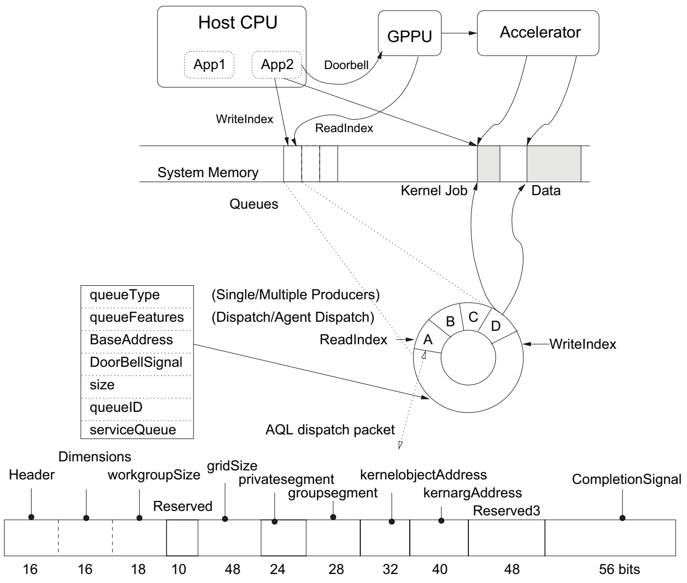

Fig. 3. Dispatching a job to an accelerator via the GPPU; the GPPU monitors both the accelerator progress and the flow of submitted jobs. Queue context and packet format layout; packet size is 64 bytes, fields order to transfer a packet is from the 'Header' field

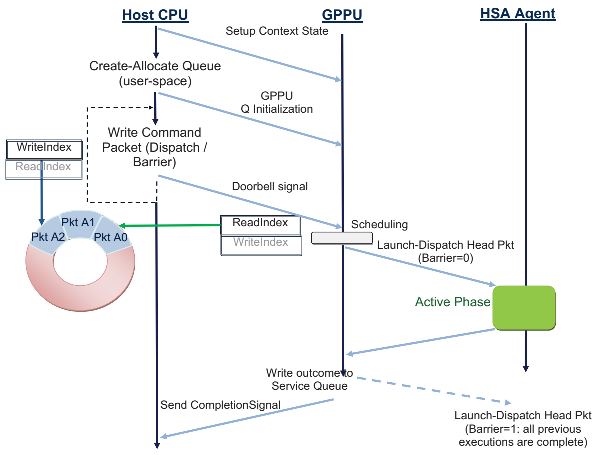

Fig. 4. Communication protocol in dispatching a task to HSA Agent through the GPPU mediation

user application is then allowed to enqueue command packets to the ring buffer using the Packet ID information. In fact, a new Packet ID is obtained by calling the GPPU runtime through a specific API. Thus, by acquiring the new packet ID an application can calculate the virtual address (*userVA*) to find the available packet within the ring buffer. Using the userVA the packet can be populated according to the predefined AQL format including parameters and pointers to the workset. Finally, the application creates a signal to monitor the task completion and notifies the GPPU that the packet is ready to be processed via a *doorbell*. The doorbell signaling allows each application that offloads packets to notify the GPPU of packets that are ready waiting to be served. *Signals* can be actually considered shared memory locations containing an integer. The GPPU will dispatch all packets from a circular queue until a barrier packet is identified. All packets that have been dispatched must reach their completion phase before any other packet from the same queue will be launched.

During the GPPU dispatch phase, when the doorbell has been received, the physical address (PA) of the AQL packet is obtained using the *base_address* and the *readIndex*. At this point the packet is processed by the GPPU on the basis of its type. When the accelerator has completed processing, the GPPU can update the readIndex, and can set the packet as invalid. Finally, the signal included in the packet will be reset by the GPPU, so the host application can be waked up and carry on its execution. The GPPU may include a DMA engine in order to facilitate fast transfer of packets that are ready to be processed. Hence, a number of packets can be transferred to the GPPU's local buffers for processing ahead of time while the accelerator is active.

IV. Design and Realization

A proof-of-concept design is prototyped utilizing the ZYNQ SoC architecture [28], which includes two ARM Cortex A9 CPUs with 32KB I/D L1 cache, and 512KB L2 cache, while the board (Zedboard) integrates also a 512MB DDR3 memory. Figure 5 depicts the organization of the System-on-Chip that includes one GPPU which interfaces two MicroBlaze soft-CPUs that act as dedicated on chip hardware programmable accelerators. We restricted the OS to use less physical memory than the 512MB of the actual DDR3 chip and the remaining physical partition is utilized to maintain the AQL data structures (the circular queues), as well as the data and kernels to be offloaded to the hardware accelerators. As shown in figure 5, the physical partition at high system memory allows the GPPU and the accelerators to access this partition without an IOMMU.

The register block is logically included inside the GPPU hardware component and is the memory-mapped interface to the AQLSM runtime. To avoid synchronization issues some registers must be exposed as read-only to the appropriate master. For instance, the queue tail pointers (writeIndex) are read-only by the GPPU, while the head pointers (readIndex) are read-only by the AQLSM.

A. Software API and Communication Protocol

The applications running on the host CPU (ARM Cortex A9) first initialize the GPPU runtime, named hereafter AQL-aware system manager (AQLSM), via *aqlsm_init()* function. Then, each application creates a queue via *aqlsm_queue_create( )* of the AQLSM, that returns a (userVA) pointer to the queue context shown in figure 3. This context will be used to access the AQL packets as described hereafter using the MMU of the Host CPU. The GPPU on the other hand accesses the same objects i.e., *queues*, *kernels* and *data*, by using the physical addresses formed by adding the memory partition's base address and the corresponding offset.

When an application calls the *aqlsm_queue_create( )* function, AQLSM delivers the *QID*, that is a UserVA pointer based

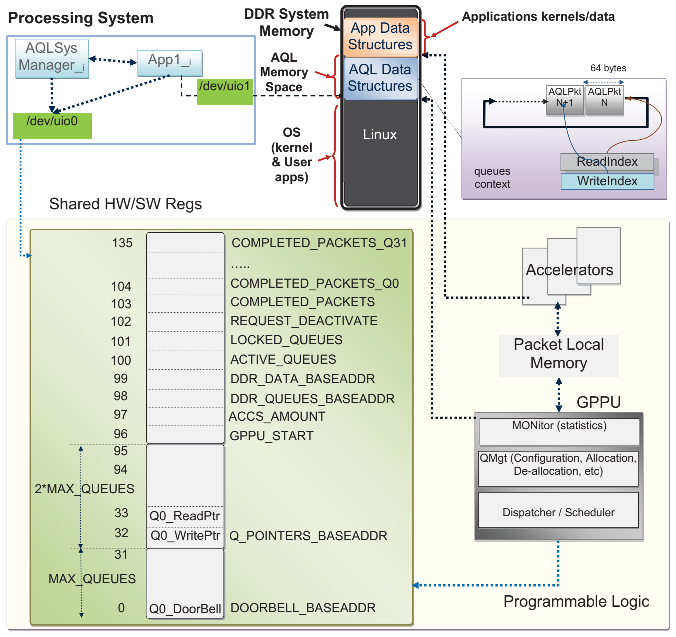

Fig. 5. Design of a GPPU subsystem on a ZYNQ SoC prototype platform and HW/SW interface registers. AQL Data Structures can also be realized in the GPPU if trading speed for scalability

on which the application accesses the following data structures directly or by calling a specific function of the AQLSM: (i) the ring buffer, which resides in system memory; the queue size is fixed and is determined when it is allocated, (ii) the data buffer (workset), which is a fixed 8KB-size partition in system memory that is private to this application, (iii) the Doorbell signal for each queue, which is located in the GPPU, (iv) the queue writeIndex and readIndex pointers to the circular buffer; the application uses the writeIndex to access the circular queue and inject a packet and the readIndex which is a shadow copy of the same register manipulated by the GPPU, in order to identify if the queue is empty or full.

In the dispatch phase the application needs to fill the contents of the packet, to enqueue the packet to the assigned queue and to signal the Doorbell notification. This is done by the AQLSM runtime API that implements the appropriate acquire release semantic. The completion phase of the acceleration process is implemented by signals stored in the AQL packets that are manipulated only by the AQLSM runtime to avoid potential attacks and misused signaling. The advantage of signal over shared memory is that it is more efficient in term of power and speed enabling each process to go to sleep and allow the AQLSM to trigger the activity of each application when the acceleration is complete and the energy constraints allow to wake up suspended tasks.

A single circular queue requires one write and one read pointer. When an application is enabled to use such a queue then a master write pointer is created and directly manipulated by the AQLSM, while a *shadow* write pointer is also created in the GPPU. At the same time a master read pointer is created and manipulated by the GPPU and its shadow read pointer is used by the AQLSM runtime. When the user application needs to enqueue one packet then the current write pointer is used and always compared to the value of the read pointer to track if the queue is full; in this case it is required to retrieve the value from the master copy of the read pointer that is located at the GPPU. The opposite happens from the side of the GPPU.

The software runtime manager for the GPPU, AQLSM (see figure 6), is running on the host CPU and communicates with the applications to manage the hybrid (hw/sw) dispatching queues and to communicate with the GPPU for initializations and queue maintenance (e.g., activation, free queues, synchronization).

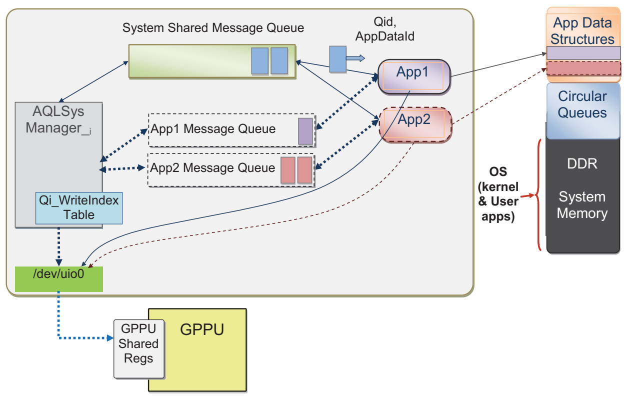

Fig. 6. The AQLSM system manager serves AQL-aware applications through SYS V IPC message queues

This AQLSM runtime provides offsets (indexes) to the GPPU and to each application. The GPPU is initialized with the base address (i.e., the physical address) of the partition of the DDR system memory that maintains the applications data and then utilizes the offsets to determine the base address of each queue.

Initially, each application receives a queue identifier (QID) after a successful allocation by the AQLSM. Then, the application uses a call to the API aqlsm_queue_add_write_index_relaxed(QID,1) of the AQLSM to request the Packet ID. Packet IDs use the write and read indexes which use queue relative values, e.g., 0, 1, 2. . .,15 for a circular buffer of 16 slots for 16 packets. Then, the application adds the Packet ID to QID (Queue pointer) to get the base address of the AQL packet. The GPPU calculates the physical address $\text{baseAddress}_{phys}$ using the base address in the AQL data structure and the pointer value times the packet size; in other words,

$$\text{baseAddress}_{phys}+((\text{QID}\times\text{queueSize})+\text{readIndex}\times\text{AQLpacketSize})$$

is the actual physical address for read. The user application on the other hand calculates the virtual address $\text{baseAddress}_{virt}$ based on the base address that is returned by the mmap function. Hence,

$$\text{baseAddress}_{virt}+((\text{QID}\times\text{queueSize})+\text{writeIndex}\times\text{AQLpacketSize})$$

is the actual virtual address used for writes, as now $\text{baseAddress}_{virt}$ is the return value of mmap.

The following algorithm depicted in Algorithm 1 listing presents the GPPU functionality.

Algorithm 1: GPPU queue management

```
input : Queues_Base@, Jobs_Base@, #Accelerators
while GPPU enabled do
    for $i \leftarrow 0$ to $MAXQ - 1$ do
        if queue active and not frozen and doorbell and available accelerators then
            update readPtr;
            fetch packet;
            assign job to accelerator;
            reduce available accelerators;
            if no more accelerators then freeze the queue;
        end
    end
    for $j \leftarrow 0$ to $MAXACC - 1$ do
        if any accelerator completed its job then
            deliver the packet to application with completion signaling;
            if readPtr equal writePtr reset doorbell;
            set the queue's complete packet status bit;
            set the packet's complete bit in queue status;
            activate frozen queue;
            increase available accelerator;
        end
    end
end
```

B. Queue Management

Queue management is performed by the AQLSM runtime which cooperates with a hardware entity that resides in the GPPU and is responsible first, to provide the pointer of the queue (QID) to each application, based (i) on the service level that is requested by the application and (ii) on the system energy and performance status and second, to communicate with each GPPU to synchronize the queue management (allocation and release) process.

Queue allocation and release techniques constitute important considerations for efficient management in terms of balancing between performance and energy constraints. GPPU supports multiple queues in order to address multiple accelerators and different priorities in dispatching. Searching for instance only active queues requires less energy with regard to the number of operations performed by the GPPU. The queue management protocol rules and operations are as follows.

- When a new queue needs to be allocated, the AQLSM reads the ACTIVE_QUEUES register of the GPPU to check which queue is empty. If the Queue is idle (deactivated), then the AQLSM first initializes the WritePtr and ReadPtr in the GPPU, then updates the ACTIVE_QUEUES register of the GPPU and finally returns the write, read pointers and the queueID to the requesting task.
- If the Queue is already active, then the AQLSM stores its deactivate command in the DEACTIVATE register in the GPPU by setting the corresponding bit in the QID that needs to release. The GPPU will serve this command when the remaining packets from the queue are dispatched and then the corresponding bit of the ACTIVE_QUEUES register will be reset.

1) Doorbell Signaling and Synchronization: The Doorbell signaling indicates to the GPPU that ready packets exist in the queue. The GPPU must examine the read and write pointers of the corresponding circular queue to identify the number of packets that are pending. After the GPPU schedules these packets it will reset the Doorbell signal if the queue read pointer reaches the write pointer. However, when the GPPU is processing the packets, one application may update the write pointer and signal the Doorbell in the meantime, thus causing a race. This new event signal may be missed when the GPPU decides to clear the Doorbell register. This race is addressed by allowing the GPPU to compare the queue write and read pointers once again, after clearing the Doorbell. If the write pointer has advanced and the Doorbell should be set then the second check will discover that a synchronization issue occurred and immediately will restore the Doorbell signal to the correct value ('1').

2) Read and Write Indexes in a Shared Queue: Each application issues a request to the AQLSM via the AQLSM_queue_add_write_index_relaxed() API. Next, the application can fill the packet fields at the received userVA and update the Doorbell signal. However, this mechanism presents potential hazards in the case of multiple applications (producers) that share the same queue. The GPPU should maintain additional pointers to handle out-of-order packet transmission since there is no guarantee that the applications will signal the doorbell in the same order as the order that the AQLSM delivers the write indexes. Thus, the GPPU could potentially advance its own readIndex past one or more packets that are not ready yet. For example, as figure 7 shows a potential scenario, two applications receive two write pointers (i.e., three and four) but may signal the doorbell and update the write pointers in a different order. Notice that the AQLSM updates the shadow pointer in the GPPU before the doorbell signal. Then the GPPU runs in risk to read the packet from a readIndex that is not valid yet; provision is mandatory if decoupling of the following events is permitted for performance reasons: writeIndex request, packet content preparation and packet launching (i.e., doorbell signaling).

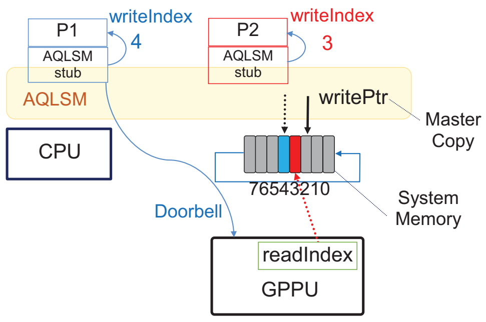

Fig. 7. Synchronization issue example; unless AQLSM allows for uncontrolled serving of application requests for available writeIndex, GPPU will access packet 3, which may not be ready yet

To avoid fairness issues or additional read pointers and complex FSMs, the AQLSM resolves out-of-order enqueues through enforcing serialization of application requests for the next available writeIndex position. One application can make a request for more than one slots and will be granted the writeIndex if no other application is pending to launch its packets. After the application initializes the packets it signals the Doorbell and notifies the AQLSM to unlock the writeIndex of this queue.

To ensure atomicity we investigated various options, mostly focused on the ARMv7 architecture. Thus, as shown in figure 8, by using the special Load-Exclusive and Store-Exclusive synchronization primitives LDREX/STREX we can achieve the lowest latency. Figure 8 demonstrates benchmarking results for a scaling number of threads that perform one million lock operations using a few different methods. Among them a custom hardware solution is shown embedded in the FPGA fabric, which costs more that the inherent ARM core atomic instructions due to the latency when the CPU accesses the hardware mutex block through a number of interconnects.

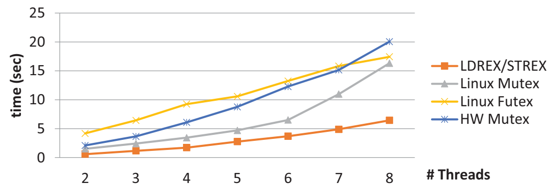

Fig. 8. Benchmarking one million atomic increments of a shared integer when scaling the number of threads from two to eight

3) Notification and Queue Deactivation Synchronization: The AQLSM decides the number of queues to activate on the basis of performance requirements and of runtime power constraints; this is among our goals in future work. Since packet processing for a particular queue may be in progress, the AQLSM issues a request for deactivation to the GPPU in order for the GPPU to complete any pending operation and then to deactivate the queue. The application that had already issued requests for this queue will be serviced and the AQLSM will not provide further access to this queue.

V. Performance Evaluation

For an efficient dispatching infrastructure in a SoC, it is imperative to minimize the costs of communication, protocol and data transfers, as well as the synchronization penalties. We have intentionally forced the MicroBlaze accelerators to execute a tiny kernel acceleration function, i.e., a matrix multiplication of 2×2 arrays of integers, in order for the computation time not to dominate the overall delay of the offloading process, and thus demonstrate the opportunities by using the proposed dispatching mechanism. Further, notice that we focus on evaluating the efficiency of the dispatching hardware and software framework and not on designing specialized circuitry for tackling the computationally-intensive functions from software and ease the burden of the processor.

As baseline system we use the developed platform with a single MicroBlaze; the AQLSM configures the GPPU to utilize only a single accelerator. Figure 9 shows the performance of the GPPU-based system when applications share a single queue or when applications utilize an independent, private queue; the plot on the left depicts the results with one accelerator and the right plot the results when we integrate two accelerators. When an application contacts the AQLSM, then it must also specify the desired type of service and in response the AQLSM delivers the appropriate queue.

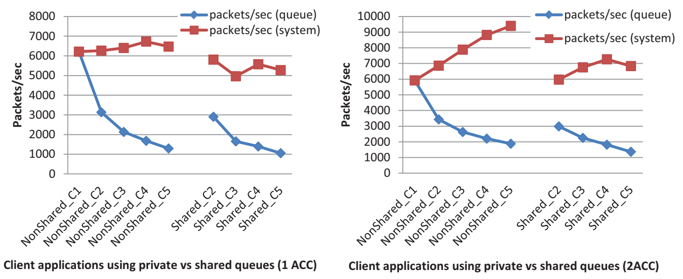

Fig. 9. GPPU performance, in packets/sec, for a single task and for the full system when dispatching tasks to HSA Agents (a single and two accelerators) using independent vs shared queues; notice that all applications (or clients) in the shared scenarios, share a single queue

A non-linear increase is observed, mainly due to the limited capacity to handle job acceleration with only two MicroBlaze cores. Additionally, contention to acquire the next available packet in a shared queue gives a foreseeable delay. Through increasing the number of queues as available resources for the applications, the GPPU can bring significant performance optimization only if we activate more accelerators. Supporting independent queues incurs queue management and synchronization, which requires negligible complexity and most importantly provides less perturbations among the different applications. When the AQLSM employs independent queues per application, then synchronization is not necessary, which results in 12.9% (2 tasks), 14.3% (3 tasks), in 17.6% and in 27.2% improvement over the shared queue case.

An application typically makes a request for an offload operation and waits for the outcome that the GPPU signals using the completion signal. However, the application can potentially submit more jobs to the accelerator engines in the meantime, if no dependencies exist. The goal is to overlap the wait interval with useful work. Figure 10 depicts the performance improvement that is accomplished when the applications submit one extra job before the previous one is completed. The rate of the processed packets shows larger improvement, almost 22%, when compared to the baseline case of a single application utilizing the accelerators.

The breakdown of a complete offload operation through the GPPU is outlined in table I. To complete a single dispatch the total latency, comprised of the GPPU operations and of the accelerator activity is 95.47 μsec. If we include the delay of the AQLSM runtime then, the delay accounts for 154 μsec in average. For reference, the time delay for the host ARM Cortex

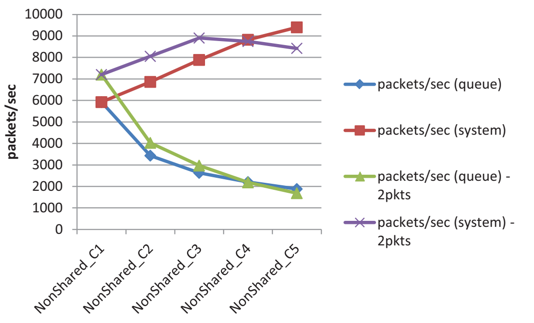

Fig. 10. GPPU performance (packets/sec) exploiting two packets(jobs) per request for a single task (queue) and for the full system (two accelerators)

A9 CPU to perform a single 2×2 matrix multiplication is 11 μsec.

TABLE I
COST BREAKDOWN OF A SINGLE JOB OFFLOAD (2X2 MATRIX MULTIPLICATION)

Operation	Clk Cycles (F: 150MHz)	Delay (μsec)
GPPU fetch/dispatch	5442	36.28
GPPU collect/send	2532	16.88
ACC processing	6347	42.31
Host-AQLSM processing (F: 666MHz)		58.83
Total processing		154.00

To analyze the sources of latency we used hardware probes (AXI performance monitors) to dynamically measure the traffic incurred by the GPPU component and by the two accelerators that are connected to different memory controller ports. Table II depicts the required throughput of the GPPU-based infrastructure to accomplish a full system performance of more than 10.5 K packets/sec when all 32 queues are active. Each application issues two requests before asserting the doorbell signal to the GPPU.

TABLE II
SUMMARY OF PERFORMANCE AND THROUGHPUT ACHIEVED BY A SINGLE GPPU TO PERFORM A 2X2 MATRIX MULTIPLICATION

Queues/Applications	1	5	16	32
Delay (512 pkt/q, sec)	0.0709	0.25	0.7769	1.547
Packets/sec (queue,2pkts)	7200.7	1775	658.9	330.9
Packets/sec (system,2pkts)	7200.7	8875	10543.9	10588.8
GPPU Throughput (MB/s)	0.924	1.28	1.35	1.36
Accelerator Thr. (MB/s)	0.52	0.722	0.759	0.762

The second line summarizes the mean latency to complete 512 dispatching operations per application. Since both the GPPU and the MicroBlaze accelerators use separate 64-bit AXI interfaces operating at 150 MHz, which are underutilized as the measurements show, the software components of our GPPU design become the sources of latency.

A. Discussion

Data transfer and launch overheads associated with offloading is in fact a critical component of the runtime of on-chip accelerator workloads in the embedded systems domain. Even when using high-end CPUs and GPUs, researchers report more than 8.2 μsec are spent to enqueue a command into the driver and to process the command [29]. To address these delays, various mechanisms are proposed, such as early execution of kernels with special hardware support as guards on memory operations [29], or using hardware assistance by DMA units [21] [30], and even collaborative kernel driver memory management [31]. Although recent GPUs attempt to unify the CPU and GPU virtual address spaces [32], and programmers can thus use the pinned host memory to make kernels directly interact with CPU memory still data transfers suffer large delays. Nevertheless, notice that applications in our proposal benefit from sharing the same physical memory with the CPU, even if the CPU creates a virtual address space for the applications. Further, recent proposals (e.g., [30]) disregard the need to support scheduling in job dispatching. Last but not least, the proposal in [30] has unavoidable overheads due to extra copies of data to GPU external memory and due to metadata maintenance, while it requires a more complicated programming model. The advantages of our proposed IOMMU-free dispatching and acceleration are also advocated by recent works such as Dashti and Fedorova [15] that investigated in detail benchmarks by using platform architecture specifics (AMD Kaveri) along with different software stacks (OpenCl 1.2, 2.0 and HSA). The performance impact of supporting unified memory in terms of translation overheads (dTLB loads and interrupts) is remarkable, essentially showing that HSA is in its infancy over the platform they used and proving the value of our innovative scheme, beating the cost of I/O memory management.

Additionally, in comparison to using programming convenience features by the GPU industry, implementations show significant performance degradation compared to programmer controlled memory management. Hence, it is required to control the page size to match the problem size in paged GPU memory, to improve performance [24]. Our proposal is free from such GPU's MMU overheads, like handling of page translation faults, which are often disregarded ([16]). We benefit from the inherent low-latency accelerator accesses to system memory, in contrast to off-chip GPUs and off-chip hardware accelerators that require several PCIe round trips and significant interaction with the host CPU not only for data transfers, but also for translating the virtual to physical addresses. Even in heterogeneous SoCs with tighter coupling of accelerators to system memory, accelerators can still directly access the applications data without paying the costs of an IOMMU.

Finally, AQLSM runtime provides only an API to the user, thus securing user operations from illegal or accidentally wrong accesses to system memory. Essentially, AQLSM is responsible for access and sharing the GPPU queues and data partitions in a secure and fair manner. Additionally, GPPU not only dispatches job requests but also validates requests to access both accelerator and memory through queue allocation and usage control for the lifetime of the acceleration process.

VI. Conclusion

To address the challenge of efficient communication in heterogeneous architectures, we proposed an architected dispatch mechanism over unified system memory among CPUs and tightly-coupled accelerators, without the need for I/O memory management for virtual to physical address translation. By relieving the programmer from explicit management of transfers between the CPU memory and accelerators memory, we designed a scalable communication mechanism using a shared high-bandwidth memory system. We presented a realization of hardware/software dispatching infrastructure using the ZYNQ SoC that achieves more than ten thousand offload operations per second. Overall, the presented mechanisms offer substantial improvements in homogenizing the offloading process to different accelerators while eliminating overheads regarding IOMMU associated operations.

To the best of our knowledge, GPPU infrastructure is the first work to utilize packet-based dispatching while using unified physical and virtual memory between CPU and hardware accelerator, which targets high performance and easy programmability. Fitting our mechanisms with existing programming environments such as OpenCL, is among our future goals, along with integrating the GPPU infrastructure with accelerators over PCIe.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on this work. This work has been partially funded by National Matching Funds 2017-2018 of the Greek Government, and more specifically by the General Secretariat for Research and Technology (GSRT), related to EU project ”SAVE (GA No 610996), Self-Adaptive Virtualisation-Aware High-Performance/Low-Energy Heterogeneous System Architectures 01/09/2013 31/08/2016” (No 80518).

References

[1] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, “An asymmetric distributed shared memory model for heterogeneous parallel systems,” in *Proceedings of the 15th ASPLOS*, 2010, pp. 347–358.

[2] M. Coppola, B. Falsafi, J. Goodacre, and G. Kornaros, “From embedded multi-core SoCs to scale-out processors,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 947–951. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485288.2485516

[3] L. Seiler and et al., “Larrabee: A many-core x86 architecture for visual computing,” *IEEE Micro*, vol. 29, pp. 10–21, 2009.

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[5] J. E. Stone *et al.*, “Accelerating molecular modeling applications with graphics processors,” *J Comp. Chemistry*, vol. 28, pp. 2618–2640, 2007.

[6] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, “Performance gaps between OpenMP and OpenCL for multi-core CPUs,” in *Proceedings of the 2012 41st ICPP Workshops*, 2012, pp. 116–125.

[7] NVIDIA, http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2013, unified Memory in CUDA 6.

[8] V. Garca-Flores, E. Ayguade, and A. J. Pena, “Efficient data sharing on heterogeneous systems,” in *Proceedings of the 46th International Conference on Parallel Processing (ICPP)*, 2017, pp. 121–130.

[9] HSAFoundation, “HSA platform system architecture specification,” provisional 1.0 - Ratified April 18, 2014.

[10] NVIDIA, “Developing a linux kernel module using rdma for gpudirect,” TB-06712-001 v6.5, 2014. [Online]. Available: http://docs.nvidia.com

[11] Y. Durand *et al.*, “Euroserver: Energy efficient node for european micro-servers,” 2014.

[12] B. Wile, “Coherent accelerator processor interface (CAPI) for POWER8 systems.” [Online]. Available: www-304.ibm.com/webapp/set2/sas/f/capi/home.html

[13] G. Kornaros, K. Harteros, I. Christoforakis, and M. Astrinaki, “I/O virtualization utilizing an efficient hardware system-level memory management unit,” in *2014 International Symposium on System-on-Chip (SoC)*, Oct 2014, pp. 1–4.

[14] P. Vogel, A. Kurth, J. Weinbuch, A. Marongiu, and L. Benini, “Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded socs,” *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 154:1–154:19, Sep. 2017.

[15] M. Dashti and A. Fedorova, “Analyzing memory management methods on integrated cpu-gpu systems,” in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2017, 2017, pp. 59–69.

[16] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, “TimeGraph: GPU scheduling for real-time multi-tasking environments,” in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011, pp. 2–2.

[17] K. Menychtas, K. Shen, and M. L. Scott, “Disengaged scheduling for fair, protected access to fast computational accelerators,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14, 2014, pp. 301–316.

[18] R. Willenberg and P. Chow, “A remote memory access infrastructure for global address space programming models in FPGAs,” in *Proceedings of the ACM/SIGDA FPGA*, 2013, pp. 211–220.

[19] J. Cong *et al.*, “Architecture support for accelerator-rich CMPs,” in *Proceedings of the 49th DAC*, 2012, pp. 843–849.

[20] G. Kornaros and M. Pratikakis, “VWQS: A dispatching mechanism of variable-size tasks in heterogeneous systems,” in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 196–203.

[21] F. Ji *et al.*, “DMA-assisted, intranode communication in GPU accelerated systems,” in *Proceedings of the 14th HPCC*, 2012, pp. 461–468.

[22] MulticoreWareInc, “Global memory for accelerators.” [Online]. Available: www.multicorewareinc.com/gmac.html

[23] “GMAC-2: Easy and efficient programming for cuda-based systems,” NVIDIA GPU Tech Conference GTC 2012, May 14-17, 2012. [Online]. Available: http://ccoe.ac.upc.edu/projects

[24] R. Landaverde, T. Zhang, A. Coskun, and M. Herbordt, “An investigation of unified memory access performance in CUDA,” in *Proceedings of High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[25] P. Mantovani, E. G. Cota, C. Pilato, G. D. Guglielmo, and L. P. Carloni, “Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip,” in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, Oct 2016, pp. 1–10.

[26] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, “Hetero-mark, a benchmark suite for cpu-gpu collaborative computing,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–10.

[27] Tensilica Customizable Processor IP. [Online]. Available: http://ip.cadence.com/ipportfolio/tensilica-ip

[28] Xilinx, “Zynq-7000 all programmable SoC data sheet: Overview,” 2017, dS190 (v1.11).

[29] D. Lustig and M. Martonosi, “Reducing GPU offload latency via fine-grained CPU-GPU synchronization,” in *Proceedings of the 19th HPCA*, 2013, pp. 354–365.

[30] Y. Kim, J. Lee, J.-E. Jo, and J. Kim, “GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management,” in *Proceedings of the 20th HPCA*, Feb 2014, pp. 546–557.

[31] O. Tomoutzoglou, D. Bakoyannis, G. Kornaros, and M. Coppola, “Efficient communication in heterogeneous SoCs with unified address space,” in *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2016, pp. 1–6.

[32] NVIDIA, CUDA C Programming Guide.