Efficient Communication in Heterogeneous SoCs with Unified Address Space

Othon Tomoutzoglou*, Dimitrios Bakoyannis*, George Kornaros* and Marcello Coppola†
*Informatics Engineering Dept., Technological Educational Institute of Crete
Heraklion, Crete, Greece
†STMicroelectronics, Grenoble, France

***Abstract*—As recent heterogeneous system designs integrate general purpose processors, GPUs, and other specialized accelerator devices into a single platform to provide both power and performance benefits, it is important to support efficient dispatching mechanisms in terms of performance and programmability. This work proposes models for integrating hardware accelerators with applications executing under standard operating systems on an embedded processor. Instead of using direct mapping of accelerator units to user applications, or using legacy drivers that incur communication overheads and large programming effort, we develop an abstraction layer in kernel driver which driver communicates with a custom dispatcher to interface a number of hardware accelerators. At the same time we remove the need to include IOMMUs for virtual-to-physical translation from the device side, and the need to perform copies from user to kernel space when offloading computational instensive tasks to the accelerators. We demonstrate the effectiveness of our solutions running real applications on a prototype hybrid heterogeneous System-on-Chip platform.**

***Index Terms*—Heterogeneous SoC, Accelerator Dispatcher, Processor-Accelerator Unified Address Space, IOMMU-free on-chip Communication.**

I. Introduction

As general-purpose computing effectively takes advantage of throughput-oriented processing components, such as Graphics Processing Units (GPUs), different techniques emerge to amortize heterogeneity in terms of architectural differences and programmability. Multiple processing units with their own memory may incur potentially penalty to access the data if these are not in close proximity, especially if they reside in a different address space. To tackle this challenge *fused* architectures, that is multi-core CPUs and many-core GPUs are integrated on a single chip with a shared on-chip L3 cache, to produce what AMD calls accelerated processing units (APUs) [1], providing the opportunity to leverage the high computational power of the GPUs. In addition, inspired by these architectures Heterogeneous System Architecture (HSA) Foundation [2] proposes a unified virtual address space and coherent shared memory spanning the APU, enabling user-level task queues in order to reduce offloading latency.

Today, in heterogeneous architectures, computational intensive tasks, commonly called *kernels* must be launched via a run-time system through a device driver to the co-processor, and an execution context is created within the co-processor prior to execution. The co-processor commonly use a dedicated system memory partition where the data must reside and the co-processor also provides a dedicated fast memory unit and DMA engine. Thus, co-processors programming has both memory data setup and program setup overhead through the run-time system, and unless several kernels are executed sequentially in the co-processors to hide the overhead, the setup and tear down overhead for a single kernel can exceed any benefit gain via the co-processor. In the scope of HSA memory architecture an offload operation is simplified by passing virtual memory pointers to shared data from the host to the accelerator, in the same way that shared memory parallel programs pass pointers between threads running on a CPU. Even though this increases programmability and compiler implementations for offloading process, it asks for support of dedicated hardware blocks such as Input/Output Memory Management Units (IOMMUs) to allow the accelerator to handle addresses in paged virtual memory [3].

Moreover, as acceleration devices execute outside the processor, they usually lack access to processor's virtual address space and invoking the accelerator may require pinning data in memory and translating virtual addresses in advance of launching the accelerated computation. Researchers have proposed single CPU thread to assist in prefetching data for many GPU threads [4] or channels, i.e. multi-producer/multi-consumer data queues that aggregate fine-grain work items before scheduling to the GPUs [5]. In this work we address the challenge to make the data available in an efficient way in the address space of a co-processor while minimizing unnecessary time to transfer data and virtual-to-physical translations.

In addition, system software must be heterogeneity-aware in order to leverage the power-efficiency of heterogeneous systems [6]. Researchers propose heterogeneity-aware scheduling algorithms in a hypervisor and resource partitioning to allow scalability, or disengagement of scheduling extensions from the OS scheduler. In a direction similar to Beisel et al.,[7] the scheduling principles are hidden from the application developer and thus the OS can perform global decisions based on the system utilization. Application-specific scheduling inputs still have to be provided by the application developer to incorporate applications needs. As they use a hybrid user/kernel level approach to perform heterogeneous scheduling, we also have a hybrid scheduling mechanism near the accelerator processing engines that provides a unified general-purpose interface to the system OS and applications.

The main contributions of this work are summarized next.

978-1-5090-2520-6/16/$31.00 ©2016 IEEE

- An architecture to integrate multiple fine-grain asymmetric accelerators with a common unifying interface.
- An efficient accelerator interface that allows the developed kernel driver (i) to interact and configure this interface/dispatcher, (ii) to supply multiple jobs to a runtime programmable number of buffers, thus isolating and decoupling from the particular accelerators functionality and operation rate. These buffers reside in a scratchpad memory which is tightly coupled with the accelerators in order to overlap communication with processing time.
- Disengagement of the completion phase from the launch phase for an offload operation instead of the common binding of the user application to the accelerator driver for the lifetime of the offloading processes.

The rest of the paper is organized as follows. Section II presents related work. Section III describes our framework and and its implementation on our evaluation platform, while section IV analyzes the hardware-software interface. Section V provides the description of our optimized solution and finally Section VI concludes the paper.

II. Related Work

Fixed-function accelerators commonly use scratchpad memories and DMA engines or caches. Cache-based memory hierarchy is inefficient for bulk data movement in I/O bound applications, but can be more efficient at on-demand, fine-grained data transfer for applications, and depending on the applications memory access patterns [8]. In industry, IBM introduced the Coherent Accelerator Processor Interface (CAPI) on POWER8 systems to provide a high-performance solution for implementing client-specific computation-heavy algorithms on FPGAs [9]. On top, the Multi Accelerator Platform Engine for Baseband (MAPLE-B) consists of a programmable-system-interface (PSIF) that is a programmable controller with DMA capabilities and signal-processing accelerators attached using an internal interface [10] [11]. Intel's open approach to FSB-coupled accelerators proposes the QuickAssist Technology Accelerator Abstraction Layer to simplify the use and deployment of tightly-coupled accelerators[12][13]. By using a reduced functionality or "bare-bone" API, named *dataplane API*, or by batching requests, Intel aims to reduce offload latency.

Automatically managing data and optimizing communication has been addressed in the context of CPU-GPU data management through using compiler-assisted techniques to transfer data to the appropriate memory space [14][15]. In the StarPU runtime [16], applications submit computational tasks, forming a task graph, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. StarPU optimizes data transfers using prefetching and overlapping. In the same scope semi-automatic techniques, such as Global Memory for Accelerators (GMAC) require programmers to add annotations [17]. Unified virtual addressing [18] (UVA) abstracts away from the programmer the actual location of data, whether on any of the GPUs or on the CPU. On top we propose methods to eliminate time-consuming operations in cases that can be avoided, such as in small or fine-grain data transfers.

In order to tackle offloading overheads, researchers have also proposed dynamically aggregating asynchronously produced fine-grain work into coarser-grain tasks leveraging the GPU's control processor to manage those queues [5]. Optionally, engineers develop hardware-assisted direct memory access (DMA) and the I/O read and- write access methods along with on-chip microcontrollers inside the GPU to offer effective solutions in terms of reducing the data transfer latency for concurrent data streams [19]; Fujii et al. [19] showed that direct I/O operations are faster than using DMA controllers for small data transfers.

III. Embedding Tightly-coupled Hardware Accelerators

The biggest constraint for a system extended with accelerator coprocessors is the requirement for the CPU to coordinate transfers of data and commands between the host and each co-processor's local memory. To avoid wasting performance on unnecessary coordination tasks one option is tightly coupling through instruction customization, such as NEON SIMD extensions for ARM processors. On the other hand though, customized hardware peripherals can easily offer domain-specific acceleration, with different organizations, hierarchical or parallel. We opted for interfacing with hardware fixed- or programmable accelerators while providing a unifying and low overhead programming API. While the integrated hardware SIMD pipelines offer significant performance improvements also by removing the need to copy data to a specialized accelerator, still, they can be outperformed by a customized hardware accelerator. We use an evaluation platform based on the Xilinx Zynq-7000 all Programmable SoC [20]. As shown in figure 1, the two NEON pipelines can be the bottleneck when multiple instances are launched on the processors. The equivalent instructions per clock cycle (IPC) for matrix multiplication reaches 1.4, while when offloading to an optimized hardware matrix multiplicator the average IPC is 3.44. We used the hardware performance counters to collect the number of hardware instructions that are executed in the ARM Cortex A9 CPU and we scaled the clock cycles that are needed by the hardware accelerator in the FPGA to match the frequency of the CPU processing system.

Scratchpad-based memory systems are currently the predominant memory system organization used by fixed-function accelerators. Each accelerator's local scratchpad is usually only visible to its own datapath. Figure 2 shows the organization of a system that integrates a number of fixed-function hardware accelerators, potentially heterogeneous ones. These accelerator processing units are non-preemptive. All hardware accelerators are controlled by a single dispatcher that is responsible to supply the parameters for launching one job and communicate with the system CPU. The dispatcher abstracts the diversity of the functionality of these custom accelerators by presenting a single general-purpose API to the host CPU and encapsulating the accelerators details. The dispatcher and

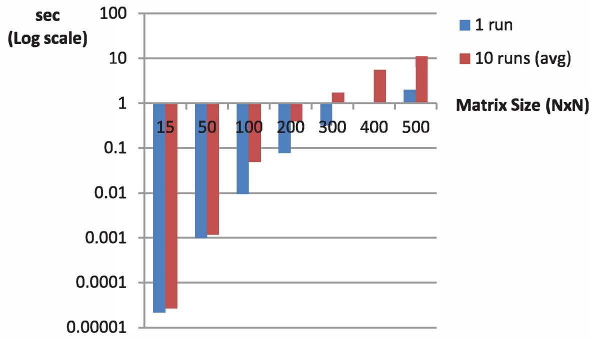

Fig. 1. Matrix multiplication for different matrix sizes using the NEON SIMD extensions on an ARM A9 processor, for a single and ten simultaneous launches

the CPU communicate through a shared scratchpad memory that stores active, launched jobs in the form of a command data structure. The dataset of the application is stored in the unified system memory, which the accelerators also access via private or shared DMA engines. However, explicit data management is not required by the programmer or by the kernel driver. Programming direct memory access (DMA) engines to copy data to/from the accelerator is done by the dispatcher. The predefined data structure that is used to exchange messages is depicted next.

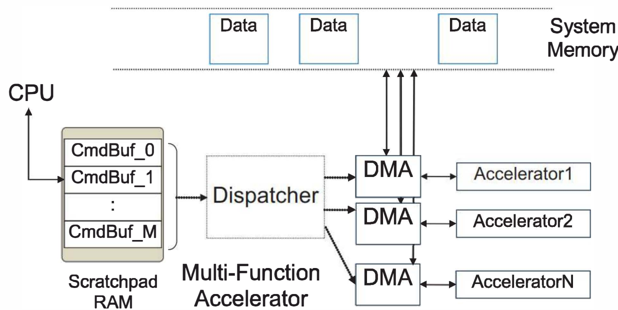

Fig. 2. Interfacing a set of accelerators through a shared dispatching controller; job contexts can be maintained in an on-chip scratchpad memory shared with the CPU, or private within the dispatcher

Arguments arg1-arg3 contain parameters of a particular function, and *ddrBaseAddr* and *result* fields contain physical DRAM addresses. We utilized field *jobs.arg0* to denote the accelerator type ID that is required to handle this job. This means that the application supplies only the type of the acceleration function that is required for the supplied data and the responsibility of scheduling this job to an available accelerator co-processor is delegated to the dispatcher. The driver is agnostic to the accelerators' functionality and is free from controlling and monitoring of each DMA engine.

In addition, control and status fields enable interaction with the overall acceleration process during operation. An application actually spin locks until the driver accepts the offload request as shown next:

```
typedef struct acc_jobs {
unsigned long const state ; /* OCCUPIED/AVAILABLE   */
unsigned long const start ;
unsigned long const status ; /* IDLE/DONE/ERROR     */
unsigned long const control ;
unsigned long queueNumber;
unsigned long packetIndex;
unsigned long long result;
unsigned long long arg0 ; /* ID-Function           */
unsigned long long arg1 ;
unsigned long long arg2 ;
unsigned long long arg3 ;
unsigned long long ddrBaseAddr ;
} __attribute__((packed, aligned(1))) jobs;
```

```
while ( io_ctl_retval == -1 ) {
 io_ctl_retval = ioctl(ramfd, CMD_WRITE_JOB, &ioctl_args_address);
}
```

and to identify the completion of the jobs as follows:

```
 io_ctl_retval = ioctl(ramfd, CMD_DONE_JOB, &ioctl_args_address);
```

IV. Hardware-Software Interface

Each accelerator can be represented as a physical device and have its own ioctl commands. However, the key idea is that customized hardware accelerators that present insignificant differences in their interface can be grouped to the same dispatcher and share the same ioctl commands. The synergy of the dispatcher with the device driver offer an abstraction layer that shields the accelerators through using a uniform data structure, *acc_jobs* and a "packetized" interface of fixed size. The dispatcher is responsible to handle the particular accelerator details. Synchronization, queuing, polling and low-level handshaking with each accelerator is assigned to the dispatcher, while the driver operates as a mediator. We differ from the bifurcated driver concept [21] that implements a fast path, in the sense that in addition to the optimized Rx/TX queue pairs split-off design we support a homogenized-unified abstraction layer.

On the programmers side, the ioctl() system calls that we developed in the form of a unified driver include: (i) SUBMIT_JOB: the first function is to copy the *job* data structure to kernel space and copy it to a dispatcher buffer, (ii) START_JOB: triggers the dispatcher to activate the accelerator by issuing a *iowrite32( )* call to set the *start* field of the job, (iii) DONE_JOB: to check the accelerator progress and identify if an error occured.

In the baseline system an application starts by deciding to offload a job to a particular hardware accelerator that is accessible through the kernel-space driver. Once the application has set up the device interface by calling the open() system call to get a file descriptor, it can access the accelerator by issuing ioctl() commands using this file descriptor. If the ioctl call is successful then a *down_write()* operation on the application's descriptor *private_data* semaphore will prevent other processes from accessing the same resource and put them to sleep. By exposing the developed ioctl command interface,

a process can submit several jobs to the available dispatcher buffers and independently start the desired job.

Notice that hardware accelerators access the same DRAM system memory that the OS uses. However, each accelerator either directly or through a DMA operation can access physical memory locations while the applications are executing in virtual address space. Since we assume an I/O memory management unit that can potentially perform virtual to physical translation is not utilized, the driver must perform these operations with phys_to_virt() and virt_to_phys() calls. Instead of translating the pages that a user application has allocated to physical addresses, (since these can be non-contiguous) a different approach is employed.

As it is commonly adopted in modern OSes, a DMA API is capable of handling coherency between CPUs and external devices when accessing the same physical memory. The driver uses dma_alloc_coherent( ) with GFP_ATOMIC flag (to avoid interrupt or SMP locks side-effects), which returns a pointer to the allocated region (in the processor's virtual address space) that is used to communicate with the user application and a "dma_handle" that is cast to an unsigned integer and given to the device as the bus address base of the region. This call ensures worry-free accesses to memory with no caching effects. However, the processor's write buffers need to be flushed before the device can read the same memory locations, and the other way round. The virtual address returned from the dma_alloc_coherent( ) call is used to perform the copy of the data from the userland by calling the copy_from_user(), as shown in figure 3. The physical address though of this kernel-space buffer is stored in the data structure that is send by using iowrite32() to the dispatcher scratchpad memory, in order for the DMA controller to work. The dispatcher FSM is then triggered to extract the appropriate fields from this command structure and program the corresponding hardware accelerator.

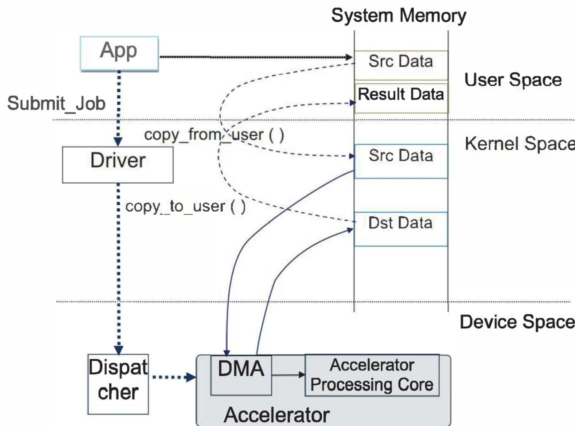

Fig. 3. Communication and phases when an application offloads a job to the hardware accelerator through a kernel driver

Instead of utilizing the DMA controller that is integrated in the processing subsystem of the Zynq-7000 family SoCs to share among the accelerators, our architecture is based on an independent DMA controller that is tightly coupled with each hardware accelerator.

The proof of concept platform is based on the ZYNQ7020 system-on-chip that is integrated on a ZedBoard platform. On top, Linux Ubuntu 14.10 (GNU/Linux 4.0.0-xilinx-11415-g4321598 armv7l) runs and we compiled test applications using gcc (Ubuntu/Linaro 4.8.1-10ubuntu8).

A. *CPU-Hardware Accelerator Communication Costs*

We quantify the cost of copy operations from user to kernel space for different matrix sizes. For accuracy we included a hardware timer attached to the dispatcher that collects the actual processing time (in clock cycles) for an offload operation. The timer measurement includes the DMA operations to get the matrices and store the result while operating in streaming mode. From the software side, the driver reported the latency of copy operations using ktime_get() (and getnstimeofday() for verification). At user level, the application used the function clock_gettime() with CLOCK_MONOTONIC option to summarize the total time of an offload operation.

Figure 4 shows the ratio of the copy operations for the two source matrices and for the resulting matrix, three in total, over the actual time required for the hardware accelerator. This is reported in clock cycles of 100MHz by using an independent hardware counter. The time values reported from the Cortex A9 CPUs are normalized to number of clock cycles (f=666Mhz) for fairness.

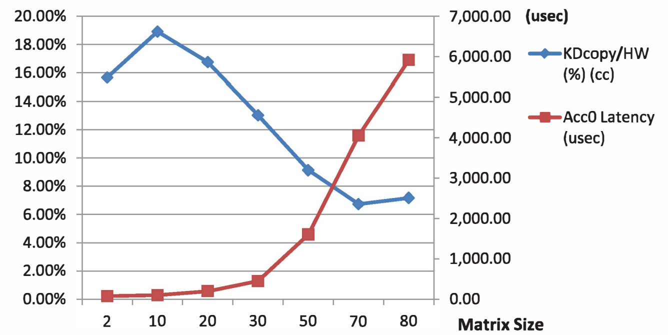

Fig. 4. Ratio of cost of copy operations over actual processing on the FPGA for matrix multiplication of varying matrix size; the vertical axis on the right depicts the performance of the hardware accelerator in μsec

Therefore, the cost of copy operations is significant, especially for fine-grain offloads. As the plot shows, in order to achieve less than 10% latency (actually 9.13%), the matrices should be at least 50×50 in size. When the hardware accelerator handles large jobs, then the ratio of the time spent in copy operations reduces. Nevertheless, this time is actually wasted, since no useful processing is done.

V. ZERO-COPY OFFLOADING

The principle to achieve an offload operation with near zero-copy operations is to design an API that allows the kernel-space driver to provide the corresponding memory area to user-space in a way that this memory space can be

physically accessed by an accelerator without any virtual-to-physical address translation. The benefits are many-fold since (i) the need for an IOMMU unit is eliminated (along with its overheads) and in the same time, (ii) copy operations are no longer necessary. Even if the dataset is larger than one page size, the risk of splitting this dataset to non-contiguous memory pages is eliminated since the allocation is done by the kernel driver, which takes care of using contiguous address space.

Initially, a driver is loaded that uses a dma_alloc_coherent() call to allocate a contiguous memory space, which is then exposed to userland by using dma_mmap_coherent(). At the same time the dma_handle is exported to the accelerator driver, since now this physical address will be used in a straightforward way.

Figure 5 shows the process for an offload operation by an application that utilizes the devices exposed by the two drivers, which we call them throughout this work as *AllocDriver* and *JDriver*. Initially, the application maps the memory area that is allocated by the *AllocDriver* to its virtual address space. This memory space is used by the application to place and initialize the matrices that need to be multiplied through using a *mmap()* system call; this system call has an average of 28usec latency in our prototype platform. Then, the application passes the same ioctl commands to the updated *JDriver*, which now does not need to perform copies; the driver now sends the job parameters directly to the accelerator and starts the accelerator. *JDriver* communicates with *AllocDriver* which exports the physical address, the *dma_handle* of the allocated-exported memory pages. Hence, what *JDriver* only does is to forward the location of the data in relation to this *dma_handle* to the accelerator. No data copy is performed in this way. When the multiplication is completed, the application is notified that the result data are ready in place. The use of atomic operations implement synchronization among applications which compete to access the device. Locking can be utilized at the kernel driver level through using *down_write()* operation, or by the applications themselves using Linux mutexes in a distributed fashion. The current driver implementation resolves contention by using the semaphore option. In addition, it returns to the application the current handle of the allocated space in order not to maintain internally data structures of the provided handles. When the offload operation completes the user provides the appropriate handle to the driver to free the space.

Figure 6 compares the acceleration in two different scenarios. The matrix multiplication benchmark is executed on the ZYNQ7020 SoC using the baseline driver (JDriver), and, in the second scenario, using the synergistic API by the *AllocDriver* and the modified *JDriver* to achieve zero-copy offloading. In the latter case the modified *JDriver* does not perform copies. Both the driver and the dispatcher use a single command buffer submitting one job at a time. As the plot depicts, the gain is larger for fine-grain jobs, bringing almost 16% advantage compared to the baseline case. Notice that the plot depicts actual collected measurements, which however involve the operation of the accelerators in 100MHz and should be scaled to the operating frequency of the processor in a real embedded SoC. By this projection of time the gain is even larger, since the actual latency of the accelerator would be much shorter.

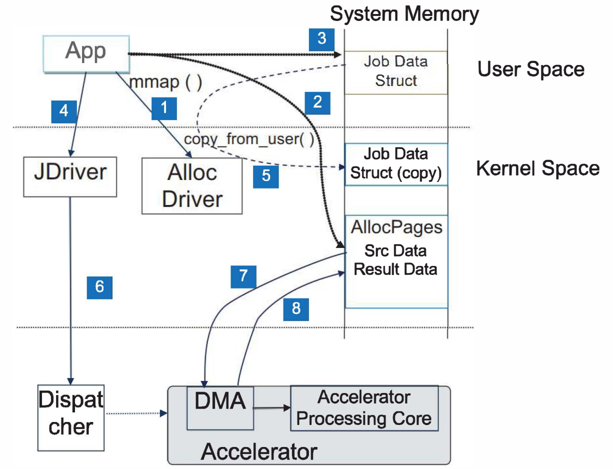

Fig. 5. Sequence of operations when an application offloads a computation through the *AllocDriver* that exports the physical memory to share with the accelerator and the *JDriver* that performs control functions using the job packet interface.

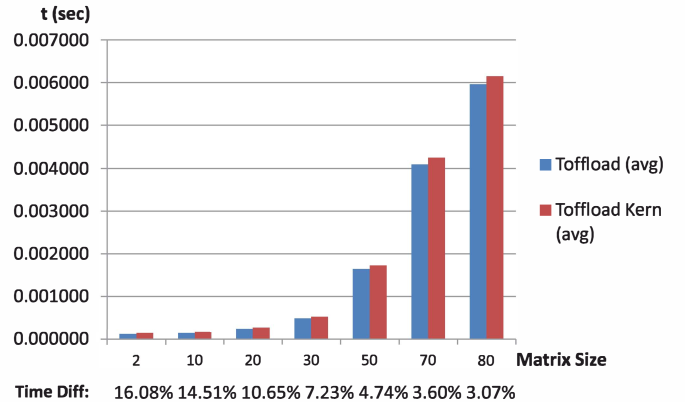

Fig. 6. Optimization results for offloading matrix multiplication for different matrix sizes (N × N), by using the exposed kernel memory to user space ($T_{offload}$), in comparison to allowing the kernel driver perform copies of data to kernel space before enabling the accelerator ($T_{offloadKern}$).

When the offloaded job exhibits large latency the percentage of the gain decreases; nonetheless, the cost of copies is still significant. The time to copy data for large matrices can amount to the total time to accelerate small jobs.

A. Optimizing Driver Dispatcher Communication

We optimized the dispatcher to use double command buffering to reduce latency when communicating to an accelerator. In addition, the JDriver now locks the job launching phase only, and not the full offload operation. Thus, another user application can also submit to the dispatcher a subsequent job that must wait in the second buffer for the active offload

TABLE I

AVERAGE LATENCY FOR OFFLOADING MATRIX MULTIPLICATION FOR DIFFERENT MATRIX SIZES (N × N), AND AVERAGE DEVIATION FROM MEAN VALUE.

Matrix Size	AVG Time (sec)	AVG Deviation
4x4	0.00015015	0.00001791
8x8	0.00016895	0.000023025
20x20	0.0003001	0.00005117
30x30	0.00052895	0.00002603
50x50	0.0852812	0.0444605
80x80	0.21362645	0.087311815

acceleration to complete. The driver now supports one more ioctl command to provide a new base address in the contiguous space if another application offload is already active. Actually, a circular data allocation scheme is implemented to simplify and provide a management scheme at low cost.

Table I shows the average deviation from the mean latency when offloading concurrently tens of user applications that offload matrix multiplication operations. When the matrices are small no significant competition is observed since the cost of a system call actually is comparable with the cost of the calculation at the accelerator. The deviation raises sharply for larger workloads.

VI. CONCLUSIONS

Even though by accessing the hardware directly enables the development of an efficient access model without the recurring overhead incurred by adding an abstraction layer, direct mapping of device resources into an application limits concurrency and schedulability. The proposed dispatching scheme is beneficial in heterogeneous SoCs with unified address space offering elimination of overheads in dynamic execution environments where offloading of many processes for short periods dominates. Our architecture can also be employed as an optimized solution in embedded systems with limited amounts of physical memory since it offers better performance predictability since other mechanisms may first need to relocate data, e.g., to move data out of the pre-allocated physical memory region, or to free space to allocate huge pages.

In the future we envision operating system interfaces that allow for virtualization of the hardware accelerator resources and their interfaces by providing both hardware and software implementations of the accelerator. Virtualization allows the operating system, guided by system policies, to share reconfigurable resources efficiently between processes while providing a consistent interface to the user.

ACKNOWLEDGMENT

This work was partially supported by the EU FP7 project SAVE under contract FP7-ICT-2013-10 No 610996. We also thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] N. Brookwood, "AMD fusion family of apus: Enabling a superior, immersive pc experience." [Online]. Available: www.amd.com/Documents/48423_fusion_whitepaper_WEB.pdf

[2] HSAFoundation, "HSA platform system architecture specification," provisional 1.0 - Ratified April 18, 2014.

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

[4] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Cpu-assisted gpgpu on fused cpu-gpu architectures," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12, 2012, pp. 1–12.

[5] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on gpus," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, 2014, pp. 181–192.

[6] A. Fedorova, V. Kumar, V. Kazempour, S. Ray, and A. Pouya, "Cypress: A scheduling infrastructure for a many-core hypervisor," in *In Proceedings of the Workshop on Managed Multi-Core Systems (MMCS'08), in conjunction with the (HPDC-17)*, 2008.

[7] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann, "Programming and scheduling model for supporting heterogeneous accelerators in linux," in *Proc. Workshop on Computer Architecture and Operating System Co-design (CAOS)*, jan 2012.

[8] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Toward Cache-Friendly Hardware Accelerators," in *Proceedings of the Sensors to Cloud Architectures Workshop (SCAW), in conjuction with HPCA 2015*, 2015.

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

[10] Freescale, "Msba8100 baseband accelerator, freescale semiconductor, inc, 2008." [Online]. Available: www.freescale.com/files/dsp/doc/fact_sheet/MSBA8100FS.pdf

[11] Analog, "ADSP-SC58x and ADSP-2158x series." [Online]. Available: www.analog.com/en/products/landing-pages/001/adsp-sc58x-adsp-2158x-series.html

[12] Intel, "Enabling consistent platform-level services for tightly coupled accelerators." [Online]. Available: www.intel.com/content/dam/doc/white-paper/quickassist-technology-aal-white-paper.pdf

[13] Intel, "Intel quickassist technology, performance optimization guide," Doc. Num.: 330687, Rev.: 1.0, Sep. 2014. [Online]. Available: https://01.org/sites/default/files/page/330687_qat_perf_opt_guide_rev_1.0.pdf

[14] T. Ramashekar and U. Bondhugula, "Automatic data allocation and buffer management for multi-gpu machines," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 60:1–60:26, Dec. 2013.

[15] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, 2012, pp. 165–174.

[16] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[17] 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," *SIGPLAN Not.*, vol. 45, no. 3, pp. 347–358, Mar. 2010.

[18] C. 2011, "Nvidia cuda programming model." [Online]. Available: http://developer.nvidia.com/object/cuda.html

[19] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for GPU computing," in *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, ser. ICPADS '13, 2013, pp. 275–282.

[20] Xilinx Inc, "Zynq-7000 all programmable SoC," Technical reference manual, Feb. 2015. [Online]. Available: www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[21] J. Ronciak, J. Fastabend, D. Zhou, M. Chen, and C. Liang, "Bringing DPDK to themainstream: The bifurcated nic driver," Mar. 2014. [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/LinuxConEurope_DPDK-2014.pdf