Direct Device-to-Device Physical Page Migrations in Multi-FPGA Shared Virtual Memory Systems

Torben Kalkhof, Andreas Koch
Embedded Systems and Applications Group
Technical University of Darmstadt
Darmstadt, Germany
{kalkhof, koch}@esa.tu-darmstadt.de

***Abstract*—Shared Virtual Memory (SVM) is a proven approach to simplify the programming of heterogeneous computing systems. It enables a single virtual address space across all computing devices, even for systems having Non-Uniform Memory Accesses (NUMA) across devices. Access time spikes due to NUMA can be reduced, though, by performing physical page migrations in SVM. These migrations ensure high data locality by moving the underlying memory pages close to the computing device currently working on the contained data, and allow the devices to fault-in pages from remote to local memories autonomously.**

The main contribution of this work is the implementation of an open-source framework enabling scalable SVM for multi-FPGA architectures, and providing efficient device-to-device page migrations. We compare the runtime of on-demand and user-managed migrations, and examine three different communication mechanisms for the actual board-to-board data transfers. Our framework supports both low-latency and high-throughput operations, requiring, e.g., only $11.6\,\mu s$ to migrate a single $4\,kB$ page between physical memories on different boards, and $760\,\mu s$ to migrate an entire $4\,MB$ range of memory.

***Index Terms*—FPGA, shared virtual memory, SVM, page migration, demand paging**

I. Introduction

The ever-growing complexity and specialization of computational problems requires more and more domain-specific solutions. Reconfigurable devices such as Field-Programmable Gate Arrays (FPGAs) are predestined to provide high flexibility for individual accelerator solutions, and are increasingly used to accelerate compute-intensive applications in both single and also multi-FPGA-settings [1], [2]. However, the resulting heterogeneous architecture of general purpose and domain-specific processing elements introduces additional challenges in programming these systems. Distinct address spaces between these processing elements are one challenge, and complicate the offloading of computations, especially if pointer-based data structures are involved. Additionally, FPGA boards offer fast locally attached memories requiring manual memory management and data movements by the programmer.

Originally developed for loosely-coupled multi-processor systems in the 80s [3], Shared Virtual Memory (SVM) considerably simplifies including domain-specific processing elements by providing a *common* virtual address space across all processing elements, thus allowing the free exchange of data pointers between user-space software and hardware accelerators. In CPU-GPU systems, SVM is already widely-used, and often combined with physical page migrations and *Demand Paging*. Here, the underlying memory pages are *moved* from host to GPU memory to provide the fastest possible access, with the GPU faulting-in the required memory pages autonomously. This completely relieves the programmer of manual memory management and data movement tasks. Due to their pointer-rich data structures, graph algorithms are a popular application for SVM [4], [5]. Furthermore, González et al. [6] have shown that SVM can improve both programmability and performance of multi-GPU systems for a wide range of scientific applications by porting parts of the multi-zone NAS benchmark [7].

Although FPGAs are an energy-efficient alternative to GPUs for many applications, a comparable framework for multi-FPGA systems is currently missing. Hence, we extend the existing SVM framework by Kalkhof et al. [8] to support *multiple* FPGA boards, and propose, to the best of our knowledge, the first SVM implementation with a direct Device-to-Device (D2D) page migration mechanism. We compare three different approaches, using both PCIe and 100G Ethernet links, to perform our direct D2D transfers.

This work is structured as follows: First, we introduce some basic terminology in Section II, and give a survey of related work in Section III. In Section IV we present the existing single-device TaPaSCo SVM feature, and propose our implemented extension for multiple devices. Finally, we evaluate our different approaches for D2D migrations in Section V, before we conclude our work in Section VI.

II. Background

In most modern Operating Systems (OSs), every user space process runs in a separate virtual address space. The OS saves the mappings between virtual and physical memory pages in multi-level page tables. Translating a virtual address by iterating over the levels of a page table is called a Page Table Walk (PTW). CPUs usually support virtual memory by providing a Memory Management Unit (MMU). The MMU is capable of performing PTWs autonomously, and caches recent translations in a Translation Lookaside Buffer (TLB). If an address is requested for which no mapping is found, the MMU

This research was funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01 IS 21007 B.

raises a *page fault*, and the OS populates the missing page, e.g., by retrieving it from disk.

In contrast, hardware accelerators implemented on FPGAs traditionally work directly on *physical* addresses to access data in memory. In case of PCIe-attached FPGA boards, such as the Alveo U280 [9], the required data is usually *copied* to on-board or on-device memory prior to launching the accelerator to benefit from higher bandwidth and lower latency of memory accesses to local memory, compared to data accesses to host memory via the PCIe bus.

However, this need for explicit data movements and the handling of distinct address spaces between user space software and a hardware accelerator complicate programming. SVM is a proven solution to simplify this task, namely by extending the virtual address space of the user process to the hardware accelerator on the FPGA. Thus, SVM relieves the programmer from performing explicit data movements or data pointer relocations, which are especially expensive on indirectly addressed dynamic data structures. SVM can be realized either with shared *physical* memory, which is the more common solution these days in the FPGA world, or *physical page migrations*, where memory pages retain their virtual address, but are moved (*migrated*) between different physical memories to be closer to specific processing elements (e.g., CPUs, GPUs, FPGAs).

III. Related Work

On embedded System-on-Chips (SoCs), CPU cores and FPGA parts often share physical memory, which is exploited by various existing SVM implementations such as [10]–[12].

In the HPC domain, shared physical memory platforms have been used to implement SVM as well. The Convey Hybrid-core machine [13] provides a host CPU and an FPGA-based co-processor, which share host and device memory pools coherently. To allow faster memory accesses, the user may move data between these two pools manually. However, there are no means for automatic migrations [14]. Ng et al. [15] propose SVM for PCIe-attached FPGA boards by providing the accelerator direct access to pages located in host memory via the PCIe bus. CAPI [16] is layered on top of PCIe as well, but allows cache-coherent accesses to host memory, in contrast to Ng et al. [15]. However, both approaches suffer under the rather high latency of accesses via PCIe. Hence, upcoming cache-coherent bus standards such as CCIX [17], CXL [18] and OpenCAPI [19] aim to achieve much lower access latency to host memory. Even closer coupling between CPU and FPGA is achieved on the Intel HARP platform by using UPI [20], and by the *Enzian* [21] research computer with its custom interconnect ECI.

Physical page migrations allow to combine the advantages of SVM with the low latency and high bandwidth of locally attached memories for off-the-shelf FPGA boards. After the required memory page is moved to more local device memory, the accelerator can access the data very efficiently. Hence, in *HelmGemm* Diamantopoulos et al. [22] extend CAPI [19] by *Demand Paging*. As soon as an accelerator on the FPGA issues a memory request, the respective memory page is moved entirely to device local memory if possible. Although *HelmGemm* uses an SVM platform providing two FPGA boards and four GPUs, it does explicitly not implement means to directly migrate pages between FPGA boards, or FPGA board and GPU.

Coyote [23] provides page migration capabilities without requiring dedicated hardware support on host side. It splits the FPGA in multiple sub-regions equipped with a separate on-FPGA TLB each. Address translations and on-demand page migrations after TLB misses are handled in software on the host. *Coyote* supports the migration of standard $4\,\mathrm{kB}$ and huge $2\,\mathrm{MB}$ pages. The framework of Kalkhof et al. [8] implements SVM with physical page migrations as well, however includes deeper OS integration by using the Linux Heterogeneous Memory Management (HMM) API [24]. The HMM API allows to manage device memory using *device private* Page Table Entrys (PTEs) together with the OS page tables. Additionally, it provides *MMU Notifiers* to keep on-device TLBs consistent, and helper functions for page migrations to and from device memory. This ensures data consistency at all time, and enables automatic back-migrations to host memory after a data access by software on a CPU causes page faults. However, the HMM API is limited to standard $4\,\mathrm{kB}$ pages. In addition to demand migrations, Kalkhof et al. [8] offer a user-managed approach, in which the programmer explicitly initiates the migration of an address range to or from device memory to allow more efficient migrations of large buffers. They integrated their framework as an additional feature into the open-source TaPaSCo framework [25], which previously just supported traditional DMA copy-based memory management.

Since both Coyote [23] and the framework of Kalkhof et al. [8] are currently limited to one FPGA board, and even the extended CAPI version of *HelmGemm* [22] does not support FPGA board-to-board migrations, we extend the TaPaSCo SVM feature of Kalkhof et al. to support *multiple* FPGA boards connected to the same host with direct migration capabilities in this work. We examine variants using the PCIe root complex on the host for Device-to-Device (D2D) transfers, as well as more scalable ones relying on 100G Ethernet links between FPGA boards to move the pages without host intervention.

SVM is already commonly used in GPU programming. It is included in CUDA (as Unified Virtual Memory (UVM)) [26] and OpenCL [27], and state-of-the-art GPUs provide hardware support in the form of powerful IOMMUs. A major benefit of GPUs is the very high available memory bandwidth of on-device memory. Hence, SVM on GPUs is usually implemented with page migrations and demand paging.

Both the AMDKFD [28] and NVIDIA UVM drivers [29] support SVM across multiple GPUs, and therefore use the Linux HMM API [24]. However, the AMDKFD driver always performs GPU-to-GPU page migrations in *two* steps. First, the pages are completely migrated to host memory, before they are migrated to the destination device in a second step. In contrast, the NVIDIA UVM driver is able to perform *direct* GPU-to-

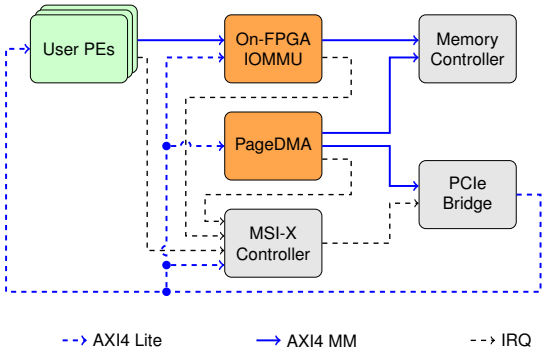

Fig. 1. Simplified hardware design of the existing TaPaSCo SVM feature. The SVM specific components are marked in orange.

GPU migrations using peer-to-peer transfers if supported by the respective GPUs.

Recent research on improving performance of SVM in multi-GPU systems covers, e.g., advanced page placement techniques [30], caching remote pages [31], or replicating pages on multiple devices [32].

IV. Implementation

In this section, we first give an overview of the existing TaPaSCo SVM feature by Kalkhof et al. [8], which we use as a baseline. Afterwards, we describe our extension of the feature to support more than one FPGA board without changes to the application hardware design, and finally present how we leverage the direct D2D migration capabilities with an additional 100G Ethernet link between the FPGA boards.

A. Existing TaPaSCo SVM Feature

Figure 1 shows the simplified hardware design of the TaPaSCo SVM feature. It uses an on-FPGA IOMMU, which is located between the user Processing Elements (PEs) and the device memory controller, to translate the virtual to physical addresses. The IOMMU has a two-level TLB structure. While the Level 1 TLB is a fully-associative LUT- and register-based design with two cycles lookup latency, the Level 2 TLB follows the set-associative BRAM-based approach of Vogel et al. [12] to achieve higher capacity than typical on-FPGA TLBs, however at the cost of variable lookup latency and lower throughput. In parallel to this two-level structure, there is a third TLB with entries of flexible length, which is used to map large virtually and physically contiguous memory regions more efficiently. The number of in-flight read and write requests is limited to 16 each, since the IOMMU has only finite capacity to track the state of every request (e.g. TLB hit/miss) until its completion. TLB misses and page faults need to be resolved in software.

The SVM feature uses a custom DMA engine, named *PageDMA*, instead of the default TaPaSCo DMA engine. It has a very simple structure, since during SVM operation only entire 4 kB pages are moved between device and host memory, which corresponds exactly to the maximum burst size of AXI4 [33]. Physically contiguous blocks of pages can be copied in-flight with only one copy command. Additionally, the core allows to initialize pages in device memory with known values. This is handy if uninitialized buffers consisting of zero-pages are migrated to the device, since no actual data needs to be copied and all initialization occurs on-device.

The page migrations are managed in the device driver. Kalkhof et al. [8] introduced two different types of page migrations, namely *On-Demand Page Migrations (ODPMs)* and *User-Managed Page Migrations (UMPMs)*. ODPMs are triggered by a page fault, either a CPU page fault on a page located in device memory, or a device page fault of the on-FPGA IOMMU after a TLB miss. Device page faults are handled in a kernel worker thread of the concurrency managed workqueue (cmwq) [34]. This worker thread handles all pending device page faults in-flight to reduce migration overhead by migrating virtually contiguous pages together. First, it performs a PTW over the CPU page table to check whether the requested pages are already located in device memory, and it is sufficient to re-add the evicted TLB entries. Afterwards, the remaining faults are sorted to allow more efficient migration of adjacent memory pages in the following steps.

In contrast to the possible in-flight handling of multiple pending faults in Host-to-Device (H2D) direction, CPU page faults must be handled one-by-one. Here, the fault handler of the Linux kernel calls the respective migration function for every page separately, and the device driver has no option to postpone a fault.

As an alternative to ODPMs, the SVM feature also offers UMPMs, where the programmer specifies virtual memory ranges which need to be migrated to device memory prior to launching the accelerator, or back to host memory after the accelerator has finished its computation respectively. This reduces the overhead compared to ODPMs significantly, as it allows to handle page migrations in larger batches, while during ODPMs the fault handling - and thus also the migration - can only be performed in small batches, or even one-by-one. Note that it is also possible to combine ODPMs and UMPMs, e.g., to handle larger known bulk transfers using UMPMs, and rely on ODPMs for less-predictable accesses (e.g., traversing dynamic data structures).

The required steps to migrate a virtual memory range are as follows. In case of Device-to-Host (D2H) migrations, the driver invalidates the corresponding TLB entries in the on-FPGA IOMMU first, and may have to wait for in-flight memory accesses to device memory by querying the IOMMU for active requests before starting the migration. Then, it collects all source pages, which need to be migrated, in a PTW using the Linux HMM API [24]. The HMM API also ensures to invalidate mappings for these pages in other TLBs in the system, e.g., in the host CPU's MMU. Next, all required memory pages are allocated in destination memory. To this end, the HMM API offers special *device private* PTEs, which are retrieved by the driver during initialization, to represent the device memory in the same manner as system memory. Now, the actual DMA transfer to copy the data from source to destination memory follows. During H2D migrations the

driver adds new TLB entries to the on-FPGA IOMMU for the migrated range next. Finally, the migration is completed by using the HMM API to update the CPU page table with the new PTEs in destination memory, and to free the source pages.

During the integration of their work into TaPaSCo [25], Kalkhof et al. improved the allocation of destination pages as follows. When migrating virtually *contiguous* pages, these are then mapped to a *physically* contiguous memory block. This reduces the number of required DMA commands, increases the efficiency of DMA copies, and guarantees that the additional TLB with flexible length entries can be used for larger mappings. For more details on the prior work, please see [8].

B. Multi-FPGA Support and Direct Device-to-device Migrations

With the existing TaPaSCo SVM feature, only one device can be used in parallel. We extend this feature here in order to allow scaling to multiple FPGA boards in a user application. Since there are more than two physical memories in the shared address space now, namely host memory and *at least two* distinct device memories, the physical pages of a virtual memory range we would like to migrate to a specific device may be distributed over *multiple source memories*. This is unfortunately no longer compatible with the Linux HMM subsystem, as the helper functions for page migrations in the HMM API [24] always expect exactly *one source* and *one destination* memory to be specified. Hence, we need to *split* the migration of a virtual address range which includes pages located in different source memories, into multiple page migrations, one for each source memory. Our changes to the framework are therefore threefold. First, we add a means to efficiently track which virtual address ranges are currently located in which memory. Second, we introduce direct D2D migrations in addition to H2D and D2H migrations. And third, we extend the migration flow to handle page migrations with multiple source memories.

1) Tracking of virtual address ranges: In principle, a PTE contains the information on which physical device the page is located. However, checking all PTEs of a virtual address range prior to starting the migration to get the respective source memories would require many additional slow PTWs. Hence, the driver stores all virtual address ranges currently located on a specific device, in one *interval tree* per-device. Since we do not store the corresponding physical addresses in these interval trees, we can keep them small by merging virtually contiguous address ranges, although the underlying physical pages may actually be discontiguous. We chose the distributed approach with *per-device* interval trees to match the modular design of TaPaSCo [25], and also to run multiple application processes, having separate per-process address-spaces, on different (subsets of) devices. Our implementation relies on the interval tree provided by the Linux kernel [35].

2) Device-to-device page migrations: Our newly implemented D2D migrations follow the steps described in Section IV-A. However, the baseline hardware design used in the prior work (Figure 1) does not support direct DMA transfers from one device to another and needs to be extended. To this end, we examine three different approaches to achieve the desired D2D capabilities, shown in Figure 2.

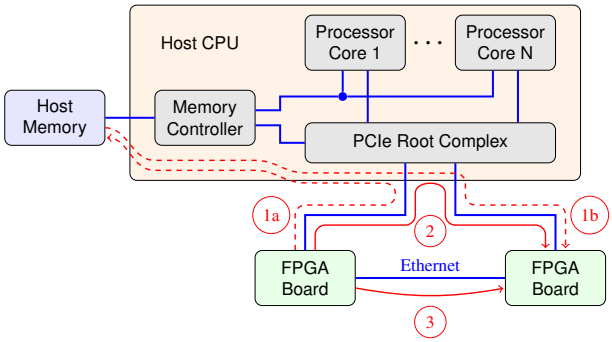

Fig. 2. Data flow for D2D transfers using PCIe and a bounce buffer in host memory (①), PCIe E2E transfers (②), and Ethernet (③).

a) Page migrations using the existing hardware design: With the existing hardware design of the TaPaSCo SVM feature, a direct copy between two devices is not possible, since the DMA engine of one device cannot accesses the second device's memory. A simple solution around this is to copy the data in two steps similar to AMD's approach. First, we copy the data from source *device* memory to a DMA buffer in *host* memory, and afterwards to destination *device* memory. Note that the AMDKFD driver [28] uses a more involved process, as it performs a complete D2H migration, followed by a H2D migration to the destination device, including page table updates in *both* steps, if a page migration from one GPU to another is required. In contrast, we streamline the process to split the actual data movement, but only update the CPU page table once, namely directly with the PTEs in destination device memory.

b) Page migrations using PCIe E2E Transfers: Obviously, splitting the data movement into separate copy steps adds additional overhead to our migration. Also, it is desirable to skip the bouncing of data through host memory. One way to achieve this are direct Endpoint-to-Endpoint (E2E) transfers using the PCIe bus. We expose the device memory to the PCIe bus using an additional Base Address Register (BAR). Hence, the DMA engine of one device can now access the memory of other devices as well, and we use the DMA engine on the source device to copy the data to destination memory.

c) Page migrations using Ethernet: Exposing gigabytes of memory to the PCIe bus using a BAR is not always desirable or even possible, e.g., if 64-bit addressing is not available, as in many mid-size embedded systems (Linux capable, but still using 32-bit addressing). Also, the PCIe E2E transfers put additional load on the PCIe root complex, which is usually used by many other devices as well. Hence, we add an additional *more direct* connection between our devices to perform the data movements independently of the host. We choose 100G Ethernet as an easily scalable solution, since we can build a low-cost network across all devices using cheap off-the-shelf switches, and use the integrated 100G subsystem

on the FPGAs [36]. We extend the *PageDMA* core so that it packages the pages to be migrated into Ethernet frames on the source device, and sends the frames to the destination device. The respective destination MAC address is provided by the driver. Also, the driver notifies the DMA engine on the destination device about how many pages are arriving, and at which address they need to be written to destination memory.

To prevent possible frame loss due to a congested receiver, we implement a flow control mechanism using the IEEE 802.3x pause frame extension [37]. As soon as the main frame buffer in the destination device is full, the flow control core advises the Ethernet core to send a priority pause frame to all connected devices. The transmitting device will then immediately stop sending more data frames until pause frames stop arriving. Already transmitted frames between the buffer overflow at the destination device and the arrival of the pause frame at the source device are buffered in an additional small overflow buffer on destination side, to prevent data loss or frame re-transmissions.

3) Extending the overall migration flow: Figure 3 shows our extended flow to migrate a virtual address range to a specific device memory. The migration is either initiated on-demand by one or more device page faults occurring, and being handled by our on-FPGA IOMMU (①), or by a UMPM (⑤). In the former case, we proceed with ② by checking whether the page is already located on the respective device, and only the TLB entry has been evicted. In contrast to the baseline SVM feature, we can use our interval tree here, and only have to perform a PTW if the page is indeed already present on the device to just retrieve the physical address (③). We then re-add the TLB entry to our IOMMU (④), and return immediately to the caller (⑧), which is the page fault handling thread in this case.

If the requested pages are not already located on the device, or the migration is initiated by the user, we need to search the interval trees of *all* devices for address ranges matching the requested range in step ⑥. Sub-ranges not found in any interval tree are located in host memory. Afterwards, we perform the corresponding D2H, H2D and D2D page migrations following the steps described in Section IV-A for all sub-ranges retrieved in step ⑥, before we finally return to the caller, which is either the page fault handling thread or the user-space API in case of a UMPM.

V. Evaluation

In this section we evaluate and compare different aspects of our variants of D2D page migrations. We start by comparing the runtime of ODPMs and UMPMs in Section V-A. In Section V-B we evaluate how the copy method and memory fragmentation affects D2D transfers, and analyze how long different steps in the migration process take in Section V-C. Section V-D concludes with a comparison of D2D transfers to the existing H2D and D2H migrations.

Our test setup consists of a Xilinx Alveo U280 [9] and a BittWare XUP-VVH card [38], which are both connected via PCIe 3.0 x16 to an AMD EPYC 7302P 16-Core processor [39]

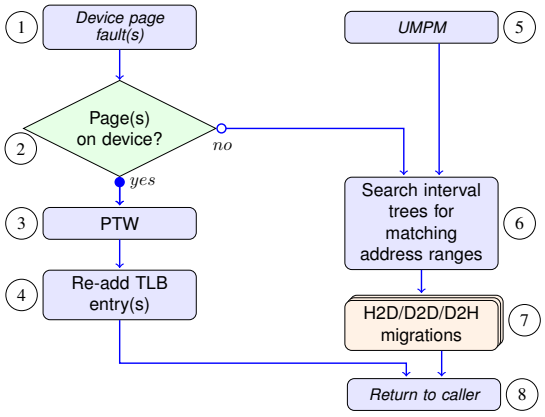

Fig. 3. Extended flow of the migration of a virtual address range. The migration is either initiated by device page faults (ODPM) or ahead-of-time by the user (UMPM).

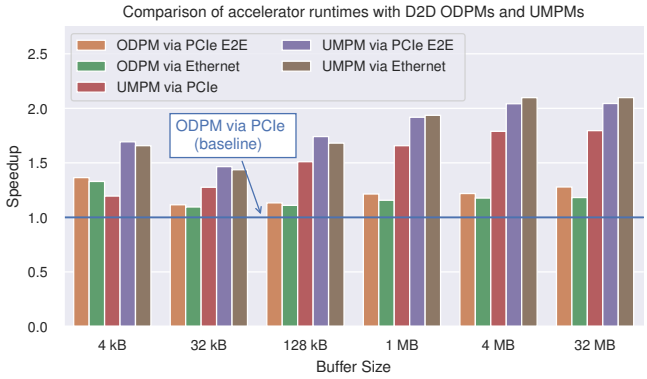

Fig. 4. Runtime comparison of the synthetic accelerator for different buffer sizes. The times include the D2D migration to move the input data to the corresponding device using ODPMs or UMPMs, and the different copy approaches. ODPMs using the two-step copy approach via PCIe and a bounce buffer in host memory are used as baseline.

running at 3 GHz and equipped with 128 GB RAM. Additionally, there is a direct 100G Ethernet connection without an intermediate switch between both FPGAs. The runtimes of D2D transfers were measured during migrations from the Alveo U280 to the XUP-VVH, while H2D and D2H transfers were measured during migrations from host to the Alveo U280, and from the XUP-VVH to host respectively. We did not observe significant deviations during runs with an opposite migration direction. Also, the measurement results are the mean values collected over 100,000 runs each for buffer sizes below 1 MB. For larger buffers, we reduced the number of runs to 10,000 or 1,000 runs, depending on the observed jitter.

A. ODPMs vs. UMPMs

In order to compare D2D transfers using ODPMs and UMPMs, we use a synthetic workload on the accelerator, which simply reads and updates all values in an array consecutively. We measure the runtime from launching the TaPaSCo task until its completion, without a back migration of data to host memory. The accelerator runtime is included, since ODPMs require the accelerator to be already running and actually cause the device page faults which trigger the page migrations. However, the runtime of the accelerator itself is

TABLE I

TOTAL RUNTIMES OF THE SYNTHETIC ACCELERATOR FOR DIFFERENT BUFFER SIZES. THE TIMES INCLUDE THE D2D MIGRATION TO MOVE THE INPUT DATA TO THE CORRESPONDING DEVICE USING ODPMS OR UMPMS, AND THE DIFFERENT COPY APPROACHES.

Type	4 kB	32 kB	128 kB	1 MB	4 MB	32 MB
ODPM (PCIe)	45.7 µs	60.4 µs	0.16 ms	1.01 ms	3.90 ms	30.3 ms
ODPM (PCIe E2E)	33.5 µs	54.2 µs	0.14 ms	0.83 ms	3.20 ms	23.7 ms
ODPM (Ethernet)	34.4 µs	55.2 µs	0.15 ms	0.87 ms	3.31 ms	25.7 ms
UMPM (PCIe)	38.2 µs	47.4 µs	0.11 ms	0.61 ms	2.18 ms	16.9 ms
UMPM (PCIe E2E)	27.0 µs	41.2 µs	0.09 ms	0.53 ms	1.91 ms	14.8 ms
UMPM (Ethernet)	27.6 µs	42.1 µs	0.10 ms	0.52 ms	1.86 ms	14.5 ms

relatively short compared the migration times and overall runtime, and we obtain a close approximation of the actual migration runtimes when using UMPMs compared to ODPMs. Note that for this round of measurements, all pages are located in *virtually and physically contiguous* memory on the source device before the migration. In Figure 4, we compare the runtimes using different buffer sizes and the three different migration methods we employ. As baseline, we always use the simplest approach, which relies on ODPMs and migrates the data in two steps by bouncing it via host memory, and show the relative speedup of the remaining variants. Absolute numbers of all runs are listed in Table I.

It is not surprising that UMPMs outperform ODPMs with all migration methods, especially for larger buffer sizes. With ODPMs, the pages must be migrated in small batches of just 16 pages each in this scenario, since this is the maximum number of read requests the IOMMU can handle in-flight and issue page faults for (see Section IV-A). This leads to increased overall migration overhead. However, even if all page faults can be handled in-flight, as it is the case for runs up to 64 kB, UMPMs are still faster than ODPMs. Here, the time required to send the page fault interrupt and schedule the fault handling worker thread is not negligible. This effect is slightly larger for 4 kB buffers due to the shorter overall runtime, and hence the relative speedup of UMPMs is higher than for 32 kB buffers. The highest speedup is achieved with UMPMs over Ethernet for buffer sizes from 4 MB on with over 2x compared to the baseline. E.g., in the 4 MB case, UMPMs speed-up from 3.90 ms to 1.86 ms. Kalkhof et al. show that, depending on the application, the use of ODPMs may be beneficial, nonetheless [8]. Also, the implementation of algorithms such as traversing pointer-based dynamic data structures is considerably simplified using ODPMs, as the designer does not have to manually orchestrate the data transfers.

Independent of the used *type* of page migrations, we achieve lower runtimes with the direct data migration methods, namely PCIe E2E transfers, and transfers via 100G Ethernet, than with two-step copies via a bounce buffer in host memory. For UMPMs, PCIe E2E transfers are slightly faster than Ethernet transfers for *small* buffer sizes. However, with increasing buffer size, Ethernet transfers achieve a higher speedup. In contrast, PCIe E2E transfers perform better for all buffer sizes when using ODPMs, since here the pages are always migrated in multiple batches, each with a maximum size of 64 kB (16 pages).

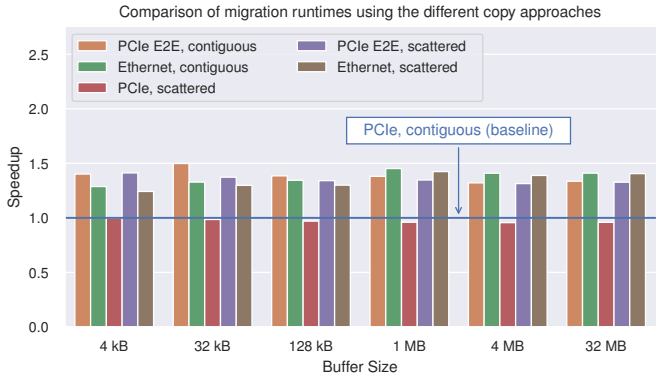

Fig. 5. Runtime comparison of D2D buffer migrations for different buffer sizes using the different migration approaches, and measured during UMPMs. The source pages may be contiguous or scattered in source memory. The migration of contiguous pages using the two-step copy approach via PCIe and a bounce buffer in host memory is used as baseline.

TABLE II

RUNTIMES OF A SINGLE D2D BUFFER MIGRATION MEASURED DURING A UMPM USING THE TWO-STEP COPY APPROACH VIA PCIE AND A BOUNCE BUFFER IN HOST MEMORY, PCIE E2E TRANSFERS, OR ETHERNET TRANSFERS. ALL PAGES ARE LOCATED IN A PHYSICALLY CONTIGUOUS MEMORY BLOCK IN SOURCE MEMORY.

Approach	4 kB	32 kB	128 kB	1 MB	4 MB	32 MB
PCIe	16.3 µs	24.6 µs	50.4 µs	0.31 ms	1.08 ms	8.42 ms
PCIe E2E	11.6 µs	16.4 µs	36.4 µs	0.22 ms	0.81 ms	6.30 ms
Ethernet	12.7 µs	18.5 µs	37.5 µs	0.21 ms	0.76 ms	5.97 ms

In general, the measured times of runs with small buffer sizes up to 128 kB are subject to significant jitter, which is due to interrupt handling. To reduce this jitter, we use *polling* instead of a interrupt-based signalling for buffer sizes smaller than 512 kB while waiting for the DMA transfer to finish. Even with this optimized DMA polling, the runtime always includes the page fault interrupt, after which the fault handling thread needs to be scheduled, as well as the PE interrupt which signals the completed computation of the accelerator, and wakes up the user application. Hence, the results reported here still have some jitter, even though we already take the mean of up to 100,000 measurements. This does not affect the ranking of the different migration mechanisms shown in Figure 4, though. Furthermore, the results may vary slightly if *more* than two FPGAs are used, as e.g., a switch in the Ethernet network could add latency to the data transfers.

B. Effect of Copy Method and Memory Fragmentation

Figure 5 compares the runtimes of D2D page migrations using the three different migration approaches and different buffer sizes. It contrasts the extreme cases how pages may be arranged in source memory: On the one hand, all pages can already be located in one *contiguous* block, and be copied with a single DMA command to the destination device. On the other hand, they may be *scattered* over the source memory, which requires individual per-page copy commands in the worst case. As baseline, we use the migration of a contiguous memory region copied in two steps via PCIe and a bounce buffer in host memory. Absolute runtimes of the runs with contiguous

TABLE III
PARTITIONING OF THE MIGRATION PROCESS IN FOUR STEPS.

Migration Step	Operations
Setup	Collect source pages, invalidate TLB(s), wait for in-flight memory accesses to complete
Allocate	Allocate destination memory/pages
DMA	Data transfer
Finalize	Update CPU page table, add TLB entries

source pages are listed in Table II. In contrast to Section V-A, we only consider the actual migration runtimes in this case, which are measured during a UMPM.

We achieve a quite constant speedup of 1.3x - 1.4x, independent of the buffer size, with direct D2D transfers, compared to the two-step bounce copy approach. The highest speedup of 1.5x is measured for PCIe E2E transfers and a buffer size of 32 kB. Direct E2E transfers reduce the migration time from 24.6 µs to 16.4 µs in this case. Conforming with the results for UMPMs in Figure 4, Figure 5 also shows PCIe E2E transfers are slightly faster than Ethernet transfers for buffer sizes below 1 MB, but slower for larger buffers.

Overall, there is no significant runtime penalty if the source pages are scattered in source memory. Only two steps of the migration process are affected by the physically scattered pages: copying the data and freeing the source memory after the successful migration. Freeing the source memory is postponed after finalizing the migration so that it does not influence the migration runtime at all. The data transfer is more complicated, since the pages must be copied one-by-one with distinct commands to our DMA engine. The driver constantly monitors the number of active requests in our DMA engine, in order to not exceed the maximum number of outstanding DMA commands. Nevertheless, there is also a slight decrease in performance for 4 kB buffers, since these buffer allocations are often unaligned to the page boundaries, and we actually have to migrate two pages.

C. Migration Step Analysis

For a more detailed analysis, we divide the migration process into the four steps listed in Table III. Figure 6 shows the relative execution time of each step for all three copy methods. Note that the behavior is very similar across all of the methods.

First, we discuss the case that all source pages are located in a *contiguous* memory block. The *Allocate* step is negligible in all cases and independent of the migrated buffer size. Also, the fraction of the *Setup* step on the total migration runtime decreases as the number of migrated pages increases. While it has a fraction of 18 - 25 % for 4 kB buffers, the fraction falls below 10 % for 1 MB buffers. In comparison to Kalkhof et al. [8], we save time during *Setup* by invalidating the entire address range in the TLB of the on-FPGA IOMMU with a *single* command, and also avoid checking for in-flight memory accesses on a per-page basis.

Most of the time is spent for the data transfer itself. In the case of direct D2D transfers over PCIe (Figure 6c) or Ethernet (Figure 6a), copying the data takes about 50 % of the total migration runtime for 4 kB buffers. After a slight decrease, we observe a jump to about 55 % at a buffer size of 1 MB, before the fraction decreases again to about 50 % for 32 MB. This jump occurs when we *switch* from polling to interrupt-based signalling for the DMA transfer to finish. For buffers smaller than 512 kB, we use polling to avoid the jitter caused by having to *wake* the waiting thread after the DMA interrupt arrives. However, for larger buffers, we do not want to block the CPU with long periods of busy waiting, and use interrupts at the cost of a slight performance penalty. Figure 6b shows a similar trend for the fraction of the *DMA* step if the two-step copy approach is used, however in the range of 60 % - 75 %. This indicates that the longer runtimes of migrations using the two-step copy approach is indeed caused by the less efficient data transfer.

With increasing buffer size, the time for the *Finalize* step is lengthening as well. However, we observe a drop at 1 MB, where we switch from polling to interrupts, and thus have longer data transfer latency, before the fraction is increasing further. For 32 MB buffers, the *Finalize* step reaches a fraction of 30 % when using the two-step copy approach, slightly over 40 % with PCIe E2E transfers, and about 45 % while using Ethernet. In this step, the HMM API [24] updates the CPU page table with the new PTEs in destination memory, and frees the source PTEs. Although we mitigate the overhead by decoupling the freeing of the PTEs from the actual memory allocation, and postponing it to after the migration, the page table update still causes more overhead with a larger number of pages to migrate.

The source pages being scattered in memory has only little effect on the fraction of the migration steps. For buffer sizes up to 32 kB, the fraction of the *Finalize* step is up to 5 % higher, which is most visible for PCIe E2E transfers in Figure 6c. Also, the *Setup* step is slightly higher for buffers in the kilobyte range in this case. Apart from that, no significant changes are observable compared to the contiguous case.

D. Comparison to Host-based Migrations

In Figure 7, we compare the runtime of D2D transfers with the runtime of H2D and D2H transfers, while using H2D transfers as the baseline. We use the results with contiguous source pages, since we have shown in Section V-B that memory fragmentation has only very little impact on the migration time. For buffer sizes smaller than 1 MB, all D2D variants are slower than H2D migrations. However, with growing buffer size, both variants with direct D2D copy transfers are *faster* than H2D transfers, with a maximum observed speedup of almost 1.4x for 32 MB and Ethernet transfers. Even the two-step copy/bounce approach over PCIe almost reaches the runtime of H2D transfers for 32 MB buffers. D2H transfers show a similar picture, however, D2D transfers outperform D2H transfers for larger buffer sizes as well.

The underlying measurement results show that the speedup of D2D transfers is *not* achieved by faster data transfers. To the contrary, the *DMA* step actually takes even *longer* than for

(a) 100G Ethernet (b) PCIe (two-step copy/bounce approach) (c) PCIe (E2E transfers)

Fig. 6. Fraction of the migration steps during D2D buffer migrations of different sizes and using the different copy methods. The source pages may be located contiguously or scattered in source memory.

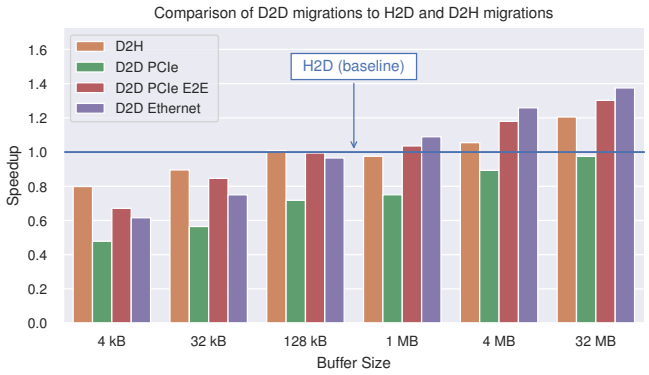

Fig. 7. Comparison of D2D buffer migrations of different sizes and using the different copy methods to H2D and D2H migrations. H2D migrations are used as baseline.

H2D and D2H migrations. However, the *Setup* and *Finalize* steps become much less time consuming for D2D migrations with increasing buffer size in comparison to H2D migrations. The speedup of D2D migrations over D2H migrations is solely achieved by lower runtimes during the *Finalize* step. These observations suggest that dealing with device private PTEs during the PTW in the *Setup* step, and page table update during the *Finalize* step is easier for the HMM API [24] than handling regular system memory PTEs. E.g. device private pages can never be pinned in memory, and are not managed by the LRU framework.

VI. Conclusion

SVM in conjunction with physical page migrations and *Demand Paging* considerably simplifies the programming of heterogeneous CPU-FPGA systems by avoiding pointer relocations and explicitly orchestrated data movements between device and host memory. Our extension to the TaPaSCo SVM feature provides SVM with *physical page migrations* across *multiple* FPGAs in parallel. To the best of our knowledge, this is the first implementation providing the means for direct D2D page migrations between two FPGA boards.

We investigated three different approaches to perform the actual data transfers during our D2D migrations: (1) a two-step approach via PCIe and a bounce buffer in host memory, which does not require any changes of the existing hardware design, (2) direct PCIe E2E transfers, by exposing device memory with an additional BAR to the PCIe bus, and (3) transfers via additional 100G Ethernet links. Our evaluation shows that direct transfers speed-up the migration by up to $50\,\%$, e.g., from $0.31\,\mathrm{ms}$ to $0.21\,\mathrm{ms}$ when migrating $1\,\mathrm{MB}$ over Ethernet. While PCIe E2E transfers are slightly faster for buffer sizes smaller than $1\,\mathrm{MB}$, Ethernet transfers perform better for larger buffers. The measured migration runtimes are nearly independent of whether the source pages are located in a contiguous block, or scattered over source memory. Using UMPMs instead of ODPMs for D2D migrations may reduce accelerator runtimes even further. Here, we achieve a speedup over 2x for buffers from $4\,\mathrm{MB}$ on with a synthetic accelerator. This is similar to the observations of Kalkhof et al. [8] for H2D and D2H migrations.

Our analysis of the different migration steps shows that despite various optimizations, the migration overhead takes up to $50\,\%$ of the overall migration runtime, especially for buffers in the megabyte range. The use of $2\,\mathrm{MB}$ huge pages could help to simplify both collecting pages in the PTW as well as updating the CPU page tables, and reduce the migration overhead further. Unfortunately, the HMM API [24] at this time only supports standard $4\,\mathrm{kB}$ pages.

In comparison to H2D migrations, our D2D migrations with direct data transfers via PCIe or Ethernet are *faster* for buffer sizes of $1\,\mathrm{MB}$ or higher, although the actual DMA transfer still takes longer. However, the handling with only device private PTEs reduces the migration overhead during the collection of the source pages and the updates of the CPU page table.

While the data transfers via Ethernet become faster than PCIe E2E transfers with increasing buffer size, we see the additional Ethernet link also as the more scalable solution. For PCIe E2E transfers, every device must expose its entire memory to the PCIe bus, which is only possible if 64 bit addressing is available. Additionally, the PCIe E2E transfers put extra load on the PCIe root complex, which is often used by many other devices as well. The Ethernet link between the FPGAs provides an additional and truly separated communication channel, and thus relieves load on the PCIe bus (and host memory banbdwidth, for the two-step copy/bounce approach). Furthermore, in a future enhancement of the framework, the Ethernet network could be used to extend the shared address space and allow page migrations to even more FPGAs that are not even directly connected to the host via PCIe.

We will publish our source code in an open-source release on Github [40].

REFERENCES

[1] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey California USA: ACM, Feb. 2017, pp. 217–226. [Online]. Available: https://dl.acm.org/doi/10.1145/3020078.3021739

[2] S. Biookaghazadeh, P. K. Ravi, and M. Zhao, "Toward Multi-FPGA Acceleration of the Neural Networks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 2, pp. 1–23, Apr. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3432816

[3] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing." *ICPP (2)*, vol. 88, p. 94, 1988.

[4] P. Wang, L. Zhang, C. Li, and M. Guo, "Excavating the Potential of GPU for Accelerating Graph Traversal," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro, Brazil: IEEE, May 2019, pp. 221–230. [Online]. Available: https://ieeexplore.ieee.org/document/8821041/

[5] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward Unified-memory-efficient High-performance Graph Processing on GPU," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 2, pp. 1–25, Jun. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3444844

[6] M. González and E. Morancho, "Multi-GPU systems and Unified Virtual Memory for scientific applications: The case of the NAS multi-zone parallel benchmarks," *Journal of Parallel and Distributed Computing*, vol. 158, pp. 138–150, Dec. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0743731521001672

[7] R. F. vanderWijngaart and J. Haopiang, "Nas parallel benchmarks, multi-zone versions," in *Supercomputing 2003*, 2003.

[8] T. Kalkhof and A. Koch, "Efficient Physical Page Migrations in Shared Virtual Memory Reconfigurable Computing Systems," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. Auckland, New Zealand: IEEE, Dec. 2021, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/document/9609831/

[9] Xilinx Inc., "Alveo U280 Data Center Accelerator Card Data Sheet," Sep. 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds963-u280.pdf

[10] H. Lange and A. Koch, "Architectures and Execution Models for Hardware/Software Compilation and Their System-Level Realization," *IEEE Transactions on Computers*, vol. 59, no. 10, pp. 1363–1377, Oct. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5374369/

[11] F. Winterstein and G. Constantinides, "Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs," in *2017 International Conference on Field Programmable Technology (ICFPT)*. Melbourne, VIC: IEEE, Dec. 2017, pp. 104–111. [Online]. Available: http://ieeexplore.ieee.org/document/8280127/

[12] P. Vogel, A. Marongiu, and L. Benini, "Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 510–525, Apr. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8519631/

[13] B. Klauer, "The Convey Hybrid-Core Architecture," in *High-Performance Computing Using FPGAs*, W. Vanderbauwhede and K. Benkrid, Eds. New York, NY: Springer New York, 2013, pp. 431–451. [Online]. Available: https://doi.org/10.1007/978-1-4614-1791-0_14

[14] Convey Computer Corporation, "Convey Programmers Guide," Nov. 2010.

[15] H.-C. Ng, Y.-M. Choi, and H. K.-H. So, "Direct virtual memory access from FPGA for high-productivity heterogeneous computing," in *2013 International Conference on Field-Programmable Technology (FPT)*. Kyoto, Japan: IEEE, Dec. 2013, pp. 458–461. [Online]. Available: http://ieeexplore.ieee.org/document/6718414/

[16] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A Coherent Accelerator Processor Interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7:1–7:7, Jan. 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7029171

[17] CCIX Consortium, "Cache Coherent Interconnect for Accelerators." [Online]. Available: https://www.ccixconsortium.com

[18] CXL Consortium, "Compute Express Link." [Online]. Available: https://www.computeexpresslink.org

[19] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 8:1–8:8, Jul. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8413085/

[20] Intel Corp. (2019, Nov.) Intel Acceleration Stack for Intel® Xeon® CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683193/current/acceleration-stack-for-cpu-with-fpgas.html

[21] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: an open, general, CPU/FPGA platform for systems software research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne Switzerland: ACM, Feb. 2022, pp. 434–451. [Online]. Available: https://dl.acm.org/doi/10.1145/3503222.3507742

[22] D. Diamantopoulos and C. Hagleitner, "HelmGemm: Managing GPUs and FPGAs for Transprecision GEMM Workloads in Containerized Environments," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. New York, NY, USA: IEEE, Jul. 2019, pp. 71–74. [Online]. Available: https://ieeexplore.ieee.org/document/8825124/

[23] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 991–1010. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/roscoe

[24] The Linux Kernel Developement Community, "Heterogenous Memory Managemet API." [Online]. Available: https://www.kernel.org/doc/html/latest/vm/hmm.html

[25] C. Heinz, J. Hofmann, J. Korinth, L. Sommer, L. Weber, and A. Koch, "The TaPaSCo Open-Source Toolflow: for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems," *Journal of Signal Processing Systems*, vol. 93, no. 5, pp. 545–563, May 2021. [Online]. Available: https://link.springer.com/10.1007/s11265-021-01640-8

[26] NVIDIA Corp., "Unified Memory Programming." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

[27] Intel Corp., "OpenCL 2.0 Shared Virtual Memory Overview." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/opencl-20-shared-virtual-memory-overview.html

[28] The Linux Kernel Developement Community, "AMDKFD Driver Source Code (version 5.16)." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/gpu/drm/amd/amdkfd?h=v5.16.15

[29] NVIDIA Corp., "NVIDIA UVM Driver Source Code (version 510.47)." [Online]. Available: https://www.nvidia.com/Download/index.aspx?lang=en-us

[30] T. Baruah, Y. Sun, A. T. Dincer, S. A. Mojumder, J. L. Abellan, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. San Diego, CA, USA: IEEE, Feb. 2020, pp. 596–609. [Online]. Available: https://ieeexplore.ieee.org/document/9065453/

[31] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka: IEEE, Oct. 2018, pp. 339–351. [Online]. Available: https://ieeexplore.ieee.org/document/8574552/

[32] H. Muthukrishnan, D. Lustig, D. Nellans, and T. Wenisch, "GPS: A Global Publish-Subscribe Model for Multi-GPU Memory Management," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Virtual Event Greece: ACM, Oct. 2021, pp. 46–58. [Online]. Available: https://dl.acm.org/doi/10.1145/3466752.3480088

[33] ARM Ltd., "AMBA AXI and ACE Protocol Specification," Jan. 2021. [Online]. Available: https://documentation-service.arm.com/static/602a9df190ee6824a1e02b98

[34] T. Heo and F. Mickler, "Concurrency Managed Workqueue," Sep. 2010. [Online]. Available: https://www.kernel.org/doc/html/latest/core-api/workqueue.html

[35] The Linux Kernel Developement Community, "Interval Tree Header." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/interval_tree.h?h=v5.16.15
[36] Xilinx Inc., "https://docs.xilinx.com/v/u/en-US/pg203-cmac-usplus," Feb. 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg203-cmac-usplus
[37] "IEEE Standard for Ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, Aug. 2018.
[38] BittWare, "BittWare XUP-VVH Data Sheet," Oct. 2020. [Online]. Available: https://www.bittware.com/files/ds-xup-vvh.pdf
[39] Advanced Micro Devices, Inc, "AMD EPYC 7302P Product Overview." [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-7302p
[40] Embedded Systems and Applications Group, TU Darmstadt, "Tapasco on Github." [Online]. Available: https://github.com/esa-tu-darmstadt/tapasco