Cluster Computing (2020) 23:1211–1234
https://doi.org/10.1007/s10586-019-02988-0

Flexible device compositions and dynamic resource sharing in PCIe interconnected clusters using Device Lending

Jonas Markussen[1,2] · Lars Bjørlykke Kristiansen[1] · Rune Johan Borgli[2,3] · Håkon Kvale Stensland[2,3] · Friedrich Seifert[1] · Michael Riegler[3,4] · Carsten Griwodz[2,3] · Pål Halvorsen[3,4]

Received: 29 March 2019 / Revised: 9 July 2019 / Accepted: 16 September 2019 / Published online: 21 September 2019

Abstract
Modern workloads often exceed the processing and I/O capabilities provided by resource virtualization, requiring direct access to the physical hardware in order to reduce latency and computing overhead. For computers interconnected in a cluser, access to remote hardware resources often requires facilitation both in hardware and specialized drivers with virtualization support. This limits the availability of resources to specific devices and drivers that are supported by the virtualization technology being used, as well as what the interconnection technology supports. For PCI Express (PCIe) clusters, we have previously proposed Device Lending as a solution for enabling direct low latency access to remote devices. The method has extremely low computing overhead, and does not require any application- or device-specific distribution mechanisms. Any PCIe device, such as network cards disks, and GPUs, can easily be shared among the connected hosts. In this work, we have extended our solution with support for a virtual machine (VM) hypervisor. Physical remote devices can be "passed through" to VM guests, enabling direct access to physical resources while still retaining the flexibility of virtualization. Additionally, we have also implemented multi-device support, enabling shortest-path peer-to-peer transfers between remote devices residing in different hosts.Our experimental results prove that multiple remote devices can be used, achieving bandwidth and latency close to native PCIe, and without requiring any additional support in device drivers. I/O intensive workloads run seamlessly using both local and remote resources. With our added VM and multi-device support, Device Lending offers highly customizable configurations of remote devices that can be dynamically reassigned and shared to optimize resource utilization, thus enabling a flexible composable I/O infrastructure for VMs as well as bare-metal machines.

Keywords Resource sharing · KVM · Composable infrastructure · Virtual machines · PCIe · Non-transparent bridging

1 Introduction

The demand for processing power and I/O resources in a cluster may, to a large degree, vary over time. Workloads come and go, and even vary themselves with number of users and amount of data to process. In this respect, efficient and dynamic resource sharing and configuration is important as it is desirable to be able to scale up and allocate more resources on demand, or scale down and release them when the resources are no longer needed. Dynamically scaling up or down based on current workload requirements, and being able to partitioning available physical resources, leads to more efficient utilization in the cluster.

VM hypervisors scale resources through device virtualization. Software-emulated devices appear to the VM guest as an I/O device, but all functionality is handled in the VM implementation. Paravirtualized devices also offer device functionality in software, but the software-defined device is backed by hardware and often resemble the physical device closely. As both methods of resource

✉ Jonas Markussen
jonassm@dolphinics.com

1 Dolphin Interconnect Solutions, Oslo, Norway
2 Simula Research Laboratory, Oslo, Norway
3 University of Oslo, Oslo, Norway
4 Simula Metropolitan Center for Digital Engineering, Oslo, Norway

virtualization require facilitation in the hypervisor, the availability of different types of resources is limited by the underlying virtualization technology being used. Furthermore, workloads that rely on multi-device interoperability become a challenge, as setting up necessary memory mappings for Remote Direct Memory Access (RDMA) and direct access between devices is generally not possible without extensive facilitation in both the hypervisor and interconnection technology. In many cases, RDMA functionality for paravirtualized devices even requires support in the VM guest drivers themselves.

In this context, a processor's I/O Memory Management Unit (IOMMU) enables devices to be *passed through* to a VM instance. A hypervisor can facilitate direct access to hardware without compromising the memory encapsulation provided by the virtualized environment. While pass-through allows physical hardware to be used with minimal software overhead, this technique does not have the flexibility of resource virtualization. Using pass-through, VM instances become tightly coupled with the physical resources they use; distributing VMs across hosts in a cluster in a way that maximizes utilization becomes a challenge.

For clusters of machines interconnected with PCI Express (PCIe), we propose a different strategy to efficient resource sharing called Device Lending [1, 2]. In these clusters, I/O devices and interconnection technology are attached to the same PCIe fabric. Device Lending exploits the memory addressing capabilities inherent in PCIe in order to decouple devices from the hosts they physically reside in, without requiring any application- or device-specific distribution mechanisms. This decoupling allows a remote resource to be used by any machine in the cluster as if it is locally installed, without requiring any modifications to device drivers or application software. However, our previous implementation lacked support for dynamically discovering the guest physical memory layout. Because of this, it was necessary to limit the VM guest's available memory in order to force certain addresses used for device memory.

In this paper, we have extended our Linux Kernel-based virtual machine (KVM) support from [2] with a mechanism for probing the memory used by the VM guest in order to dynamically detect the guest physical memory layout. This makes it possible to map device memory regions for other pass-through devices, without requiring any manual configuration of the VM instance. Such devices can then access each other, using PCIe peer-to-peer transactions. With this kind of virtualization support, it is possible to enable custom configurations of multiple devices that are passed through to VMs and enabling fast data transfers between them. In addition, we have also implemented full interrupt support, something that was missing in our previous implementation.

We present our experimental performance evaluations of multi-device configurations using GPUs and enabling peer-to-peer between them, and compare our results to bare-metal experiments. Our findings depict that we are able to borrow and use multiple remote devices, achieving the same bandwidth as native PCIe and without adding any additional latency beyond that of the interconnect and the hardware address translation. We also evaluate the performance impact of increasing the distance between devices and CPUs, particularly focusing on the impact of I/O address virtualization. Finally, we present the applicability of using the system for a realistic I/O-intensive workload, i.e., running medical image classification via deep neural networks using remote GPUs and a remote NVMe drive. We can observe that the system makes bare-metal remote execution as efficient as local execution. Our results demonstrate that Device Lending offers a highly flexible I/O infrastructure in a PCIe cluster for both VMs and bare-metal machines, allowing dynamic compositions of local and remote I/O devices.

The remainder of this paper is organized as follows: we present essential capabilities of PCIe in Sect. 2. In Sect. 3, we discuss related work. In Sect. 4, we provide an outline of our original Device Lending implementation. We describe how we have extended Device Lending with virtualization support in Sect. 5. Section 6 describes how we have added support for borrowing devices from multiple lenders. We present our performance evaluation in Sect. 7, followed by a discussion of our findings and potential improvements in Sect. 8. Finally, we conclude the paper in Sect. 9.

2 PCIe overview

PCIe is today the most widely adopted industry standard for connecting hardware peripherals (devices) to a computer system [3]. Device memory, such as register and onboard memory is mapped into an address space shared with system memory (Fig. 1). Memory operations, such as reads and writes, are transparently routed onto the PCIe fabric, enabling a CPU to access device memory, as well as allowing devices capable of DMA to directly read and write to system memory.

PCIe uses point-to-point links, where a link consists of 1 to 16 lanes. Each lane is a full-duplex serial connection, data is striped across multiple lanes, and broader links yield higher bandwidth. The current revision, PCIe Gen3 [4], has a throughput of around 13 GB/s for a x16 link.

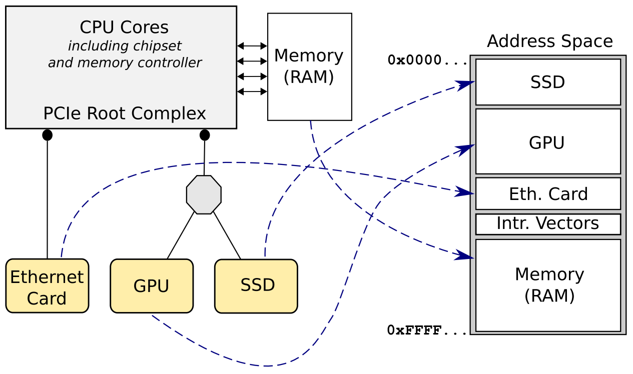


Fig. 1 Device memory is mapped into the same address space as the CPUs, allowing devices to access both system memory and other devices

Not unlike other networking technologies, PCIe also uses a layered protocol. The uppermost layer is called the transaction layer, and one of its responsibilities is to forward memory reads and writes as transaction layer packets (TLPs). It is also responsible for packet ordering, ensuring that memory operations in PCIe are strictly ordered. Underneath the transaction layer lies the data link layer and the physical layer, and their responsibilities include flow control, error correction, and signal encoding.

As shown in Fig. 2, the entire PCIe network is structured as a tree, where devices form the leaf nodes. In PCIe terminology, a device is therefore referred to as an "endpoint". Switches can be used to create subtrees in the network. The "root ports" are at the top of the tree, and act as the connection between the PCIe network and the CPU (CPU cores, chipset, and memory controller). The entire PCIe network comprises the "fabric".

Some PCIe devices may support multiple functions, which appear to the system as a group of distinct devices, each which a separate set of resources. The term "device" actually refers to an individual function. An example of a multi-function device is a multi-port Ethernet adapter, where individual ports can be implemented as a separate functions.

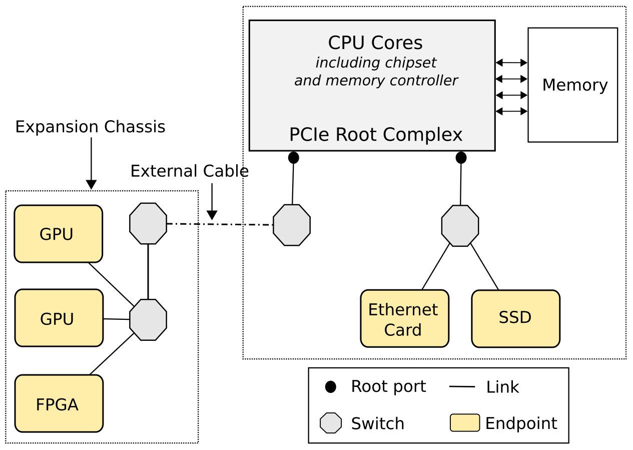


Fig. 2 Example of a PCIe topology using an external transparent link. The devices in an expansion chassis are attached to the same PCIe root as the internal devices, and are mapped into the same address space by the system

2.1 Memory addressing and forwarding

The defining feature of PCIe is that device memory and registers are mapped into the same address space as system memory (Fig. 1). Because this mapping exists, a CPU is able to read from and write to device memory regions, the same way it would read from system memory. No specialized port I/O is required. Likewise, if a device is capable of DMA, it can read from and write to system memory, as well as other devices on the fabric.

In order to map device memory regions to address ranges, the system scans the PCIe tree and accesses the configuration space of each device attached to the fabric. The configuration space describes the capabilities of the device, such as describing the device's memory regions. Switches in the topology are assigned the combined address range of their downstream devices. This allows forwarding of memory operations based on address ranges to occur in a strictly hierarchical fashion in the tree, and TLPs are forwarded either upstream or downstream. An important property of this hierarchical routing is that packets do not need to pass through the root, but can be routed using the shortest path if the chipset allows it. In Fig. 2, the internal switch in the expansion chassis is connected to the root through an external transparent link (which differs from non-transparent links). The internal switch will have the combined downstream address range of both GPUs and the FPGA, allowing TLPs to be routed directly between them without passing through the root. This is referred to as peer-to-peer in PCIe terminology.

Another significant feature of PCIe, is the use of message-signaled interrupts (MSI) instead of physical interrupt lines. MSI-capable devices post a memory write TLP to the root using a pre-determined address. The write TLP is then interpreted by the CPU, which uses the payload to raise an interrupt specified by the device. MSI-X is an extension to MSI with support for more than one address, allowing up to 2048 different, targeting specific CPUs and mandatory 64-bit addressing support.

2.2 Virtualization support and pass-through

Modern processor architectures implement IOMMUs, such as Intel VT-d [5]. The IOMMU provides a hardware virtualization layer between I/O devices and the rest of the system, including main memory. The defining feature of the IOMMU is the ability to remap addresses of DMA operations issued by any I/O device [6]. In other words, it translates virtual I/O addresses to physical addresses.

Similarly to pages mapped by an MMU for individual userspace processes, an IOMMU can group PCIe devices into IOMMU domains. As each domain has its own individual mappings, members of an IOMMU domain consequently have their own private virtual address space. Such a domain can be part of the virtualized address space of a VM, while other PCIe devices and the rest of memory remain isolated. This allows the VM to interact directly with the device using native device drivers from within the guest, while the host retains the memory isolation provided by the virtualization. This is often referred to as "pass-through".

As most device drivers make the assumption that they have exclusive control over a device, sharing a device between several VM instances requires either paravirtualization, such as Nvidia vGPUs [7], or SR-IOV [8]. SR-IOV-capable devices allow a single physical device to act as multiple virtual devices, allowing a hypervisor to map the same device to several VMs.[1]

2.3 Non-transparent bridging

Because of its high bandwidth and low latency, it is desirable to extend the PCIe fabric out of a single computer and use it for high-speed interconnection networks [9]. This can be accomplished using an NTB implementation [10]. Although not standardized, NTBs are a widely adopted solution for interconnecting independent PCIe network roots, and all NTB implementations have similar capabilities. Some processor architectures, such as recent Intel Xeon and AMD Zen, have a built-in NTB implementation [11].

Despite the name, an NTB actually appears as a PCIe endpoint. This is illustrated in Fig. 3, where the connected systems have their own NTB adapter card. Just like regular endpoints, they appear to have one or more memory regions that can be read from or written to by CPUs or other devices. Memory operations on these regions are forwarded from one PCIe network to the other. As the interconnected networks use separate address spaces, the NTB performs a hardware address translation on the TLPs during the forwarding. Consequently, NTBs create a shared memory architecture between separate systems with very low additional overhead in terms of latency.

As the address ranges associated with the NTB may be too small to cover the entire address space of the different systems, some NTBs support dividing their range into segments. A segment can be mapped anywhere into the

[1] Note that Device Lending does not make any distinction between physical devices and SR-IOV virtual devices.

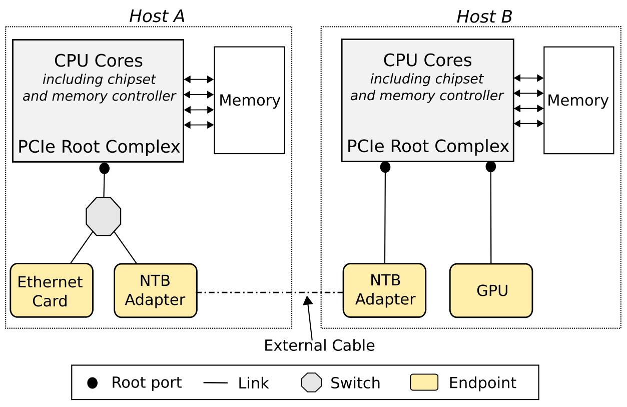


Fig. 3 Two independent networks are connected together using an NTB. The NTB Translates I/O addresses between the two different address spaces, creating a shared address space between the networks

remote system's address space. Due to the complexity of translating addresses in hardware, the number of possible mappings to remote systems is limited.

3 Related work

The idea of a unified network for the inner components of a computer with those of another is not new. It was already imagined for both ATM [12] and SCI [13]. However, these ideas never got implemented, because none of these technologies were picked up for internal I/O interconnection networks.

PCIe is the dominant standard for internal I/O bus, and is also proving to be a relevant contender for external interconnection networks. PCIe, however, was designed to be used within a single computer system only. In this section, we will discuss some solutions for sharing I/O devices between multiple hosts.

3.1 Distributed I/O using RDMA

There are several technologies which are more widely adopted for creating high-speed interconnection networks than PCIe. These include InfiniBand, as well as 10Gb and 40Gb Ethernet [14, 15]. To make use of their high throughput, they rely on RDMA [16]. Variants are summarized by Huang et al. [17] and include native RDMA over InfiniBand, Converged Enhanced Ethernet (RoCE), and Internet Wide Area RDMA Protocol (iWARP). To alleviate the complexity of programming for RDMA, middleware extensions like RDMA for MPI-2 [18] and rCUDA [19] have been developed. Those middleware extensions have also been extended with device-specific protocols like GPUDirect for RDMA [20, 21] or NVMe over Fabrics.

While RDMA extensions may achieve very high throughput on the interconnection links, they are not as closely integrated with the I/O bus fabric as PCIe, and require translation between protocol stacks. Another drawback is that it is currently only possible for such protocols to work with devices and device drivers that explicitly supports them. This is in contrast to Device Lending, which works for all PCIe devices and does not require any changes to drivers.

A proposed approach for overcoming the protocol translation overhead would be to integrate network interface functionality directly into SoCs [22], but the improvement only takes effect when the SoCs are in communication with each other. This idea is followed in the rack-scale architecture [23], which generalizes a trend returning from switched cluster architectures to hypercube architectures [24, 25]. These approaches all focus on efficient data exchange for parallel processing, rather than on resource sharing between logically separate compute units.

3.2 Virtualization approaches

Multi-Root I/O Virtualization (MR-IOV) [26] specifies how several hosts can be connected to the same PCIe fabric. The fabric is logically partitioned into separate virtual PCIe network trees, where each host sees its own hierarchy without knowing about MR-IOV. MR-IOV requires multi-root aware PCIe switches, and, in the same way as SR-IOV requires SR-IOV-aware devices to be able to provide virtual devices to several VMs, devices must be multi-root aware to provide virtual devices to several PCIe roots (and thus hosts) at the same time. Devices that are not multi-root aware can only be part of one PCIe root at the time. Despite being standardized in 2008 [26], we are not aware of any MR-IOV-capable devices. Instead, there are attempts to achieve MR-IOV-like functionality through a combination of SR-IOV with NTB-like hardware [27]. However, this approach only works for SR-IOV devices, while Device Lending makes no distinction between SR-IOV virtual devices and physical devices.

An additional virtualization approach is the Ladon system [28]. Ladon uses all PCIe and virtualization features as proposed in this paper, and is also implemented using NTBs. However, it achieves less freedom than our Device Lending, as devices are installed in a dedicated management host that manages the devices and distributes them to different remote guest VMs. In addition, devices can only be shared between different remote guest VMs, while Device Lending supports both VMs and bare-metal machines using the devices. In order to avoid management hosts becoming single points of failure, Ladon has been extended with fail-over mechanisms between management hosts in a master-slave configuration [29]. Device Lending is fully decentralized and thus avoids this all together.

Microsemi PAX [30] uses specialized PCIe switches that allow virtualization. The downstream switch ports reserve a large address range, called “synthetic endpoints”, which is similar to memory reserved by an NTB. Devices can then be hot-added through the virtual switch ports by remapping the synthetic endpoints to an actual device.

3.3 Partitioning the fabric

Rack-scale computers are so-called converged infrastructure systems, where both I/O devices and interconnects are attached to a shared PCIe fabric. Rack-scale relies on dynamically partitioning the shared fabric into different subfabrics (using fabric IDs), in order to assign individual devices to different CPUs. Unlike MR-IOV, rack-scale does not require support in devices, but it does require dedicated hardware switches which support the fabric ID header extension in order to configure routes between devices and CPUs. Additionally, these systems are only modular to the extent of typical blade server configurations, and scaling beyond a single system requires facilitation using traditional distributed methods. Adding new I/O devices requires additional modules, often only available from the same vendor.

Last but not least it should be mentioned that there have been some efforts in achieving live-partitioning using PLX PCIe switches [31], but a performance evaluation of this appears to be lacking.

4 Device lending

As illustrated in Fig. 4, it is possible to map the memory regions of remote PCIe devices using an NTB. A local CPU can perform memory operations on a remote device, such as reading from or writing to registers. Conversely, it is also possible to map local resources for the remote device, allowing it to write MSI interrupts and access the local system’s memory across the NTB.

In order to make such mappings transparent to both devices and their drivers, we have previously implemented Device Lending [1] for an unmodified Linux kernel using Dolphin’s NTB hardware and driver. Our implementation is composed of two parts, namely a “lender”, allowing a remote unit to use its device, and the “borrower” using the device. By emulating a hot-plug event [9] while the system is running, we insert a virtual device into the borrower’s local device tree, making it appear to the system and device driver as if a device was hot-added in the system. The device’s memory regions are mapped through the NTB, allowing the local driver to read and write to device

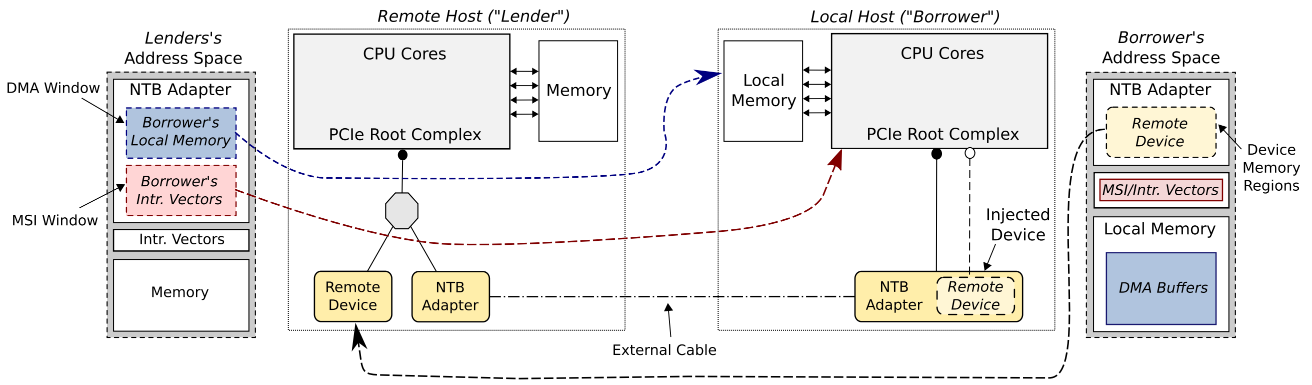


Fig. 4 Using an NTB, it is possible to map the memory regions of a remote device so local CPUs are able to read and write to device registers. The remote system can in turn reverse-map the local system's memory and CPUs for the device, making DMA and MSI possible. Device Lending injects a hot-added device into the Linux kernel device tree using these mappings

registers without being aware that the device is actually remote.

The lender is responsible for setting up reverse mappings for DMA and MSI.[2] As mentioned in Sect. 2.3, the address range of the NTB is not necessarily large enough to cover the entire address space of the borrowing system. Since it is generally not possible to know in advance which memory addresses a device driver might use for DMA transfers, we use an IOMMU on the borrower to set up dynamic mappings to arbitrary addresses, allowing the lender to set up a single DMA window. When the device driver calls the Linux DMA API in order to create DMA buffers, the borrower intercepts these calls. The borrower injects the I/O address of the DMA window prepared by the lender and sets up a local IOMMU mapping to the DMA buffer. The driver then passes the injected address to the device, completely unaware that the address is actually a far-side address. This allows the device to reach across the NTB, transparent to both driver and device. All address translations between the different address domains are done in hardware (NTB and IOMMU), meaning that we achieve native PCIe performance in the data path.

By allowing remote devices to appear to a system as if they are locally installed, Device Lending is a method for decoupling devices from the systems they physically reside in, allowing devices to be temporarily assigned and reassigned to different systems. As hosts can act as both lender and borrower, we have created a highly flexible method of sharing devices (Fig. 5). This has advantages over distributed I/O using traditional approaches; network interfaces can be assigned to a computer while it needs high throughput, and released when it is no longer needed; access latency in NVMe over Fabrics using RDMA can be eliminated by borrowing the NVMe disk instead and accessing it directly, as shown in Fig. 6; large-scale CUDA programming tasks can make use of multiple GPUs that appear to be local instead of relying on middleware such as rCUDA [19]. In contrast to RDMA solutions, Device Lending works for all standard PCIe devices, and does not require any additional support in drivers.

Our original implementation, as described in [1], did not account for peer-to-peer access when borrowing multiple devices from different lenders. As the borrowing system is not aware that the devices reside in different systems, we need a mechanism to resolve I/O addresses to other borrowed devices, in order to fully achieve device-to-device data transfers. In addition, our original implementation lacked support for borrowers that are VM guests. Adding virtualization support greatly increases the usability of Device Lending, as we introduce the flexibility of decoupled remote devices and be able to dynamically assign devices using pass-through.

5 Supporting virtual machine borrowers

Many modern architectures now implement IOMMUs, allowing DMA and interrupts to be remapped. This makes it possible for a hypervisor to grant access a driver running in a VM access to a physical device directly, without breaking out of the memory isolation, by using I/O virtual addresses. In Linux, such pass-through of devices is supported in the KVM hypervisor using the Virtual Function I/O API (VFIO) [32]. This API provides a set of functions for mapping memory for the device and control functionality, such as resetting the device, that the hypervisor can call in order to set up necessary mappings for a VM instance.

A hypothetical solution for passing through remote devices, would be for the physical host to borrow the remote device, injecting the device into its local device tree, and then implement these functions. However, this

[2] Legacy interrupts are not supported in the current Device Lending implementation, as they cannot be remapped over the NTB.

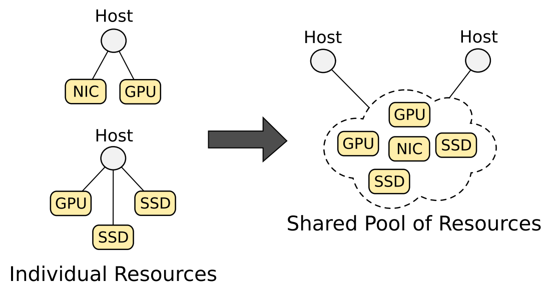


Fig. 5 Device Lending decouples I/O resources from physical hosts by allowing devices to be reassigned to hosts that currently need them. We imagine this as hosts in the cluster contributing to a shared pool of I/O resources that can be cooperatively time-shared among them

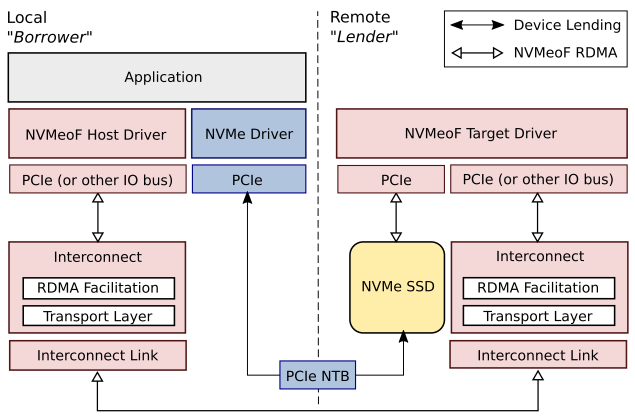


Fig. 6 Illustration of native NVMe using Device Lending compared to NVMe over Fabrics using RDMA. Device Lending makes remote devices appear as if they are locally installed and there is no need for specialized support in devices or drivers

approach would not be feasible due to the following reasons:

- The device would be borrowed by the physical host for as long as it runs, regardless of whether any VM instances would currently be using it or not. This leads to poor utilization of device resources.
- All devices borrowed by the same physical host would be placed in to the same IOMMU domain by Device Lending. VFIO requires that pass-through devices must be be placed in a per-guest IOMMU domain managed by VFIO. This is required in order to prevent memory accesses that could potentially break out of the memory isolation provided by virtualization.
- VFIO requires the entire address space of the VM to be mapped for the device. As there is no method of knowing which physical memory pages will be allocated for the VM instance before it is running, establishing this mapping in advance would require mapping all physical memory. We instead need a mechanism for only pinning and mapping the memory pages used by the VM instance in order to create necessary DMA windows.

In the 4.10 version of the Linux kernel, an extension to VFIO called mediated devices [33] was included. This extension makes it possible to use VFIO for *paravirtualized* devices. It introduces the concept of a physical parent device having virtual child devices. When a VM guest accesses the virtual device, certain operations, such as accesses to the device's configuration space or setting up interrupts, are intercepted by the mediated device parent driver. The idea is that a single physical device can be used to emulate multiple virtual devices, while still allowing some direct access to hardware. In our case, using the mediated devices extension provides us with finer grained control over what the hypervisor and guest OS is attempting to do with the device than with "plain" VFIO.

Our implementation registers an mediated device parent device for devices used by Device Lending without borrowing them first. This allows KVM to pass through the device to a VM guest without it being borrowed (and locally injected) first. Only when the guest OS boots up and resets the device, do we actually borrow the device and take exclusive control. When the guest OS releases the device, either by shutting down or because the device is hot-removed, we return the device. Not only does this limit the lifetime of a borrowed device to only when the VM is running and using the device, but it also makes it possible to hot-add a device to a live VM instance if the VM emulator supports it.

As we now have control over when a device is being used and which VM instance is using it, we can set up the appropriate isolated IOMMU groups on the lender. As shown in Fig. 7, this allows a device to be mapped in to the same virtual address space (guest-physical) as the VM as well as providing the necessary isolation to protect against rogue memory accesses. We also set up IOMMU mappings on the local system, in order to map continuous memory ranges to physically scattered memory on the host over the NTB.

While other implementations using mediated devices implement virtual child devices, each with their own set of *emulated* resources, we are passing through the *physical device itself*. This difference becomes apparent when the guest driver initiates DMA transfers; virtual device implementations emulate device registers, and are therefore able to notify KVM to pin the appropriate memory pages just before initiating the physical DMA engine. In our case, the VM instance maps the physical device registers and accesses the device directly, which means that without making assumptions about the type of device being used and implementing virtual registers for it, we are not able to replicate this specific behavior. We are also not able

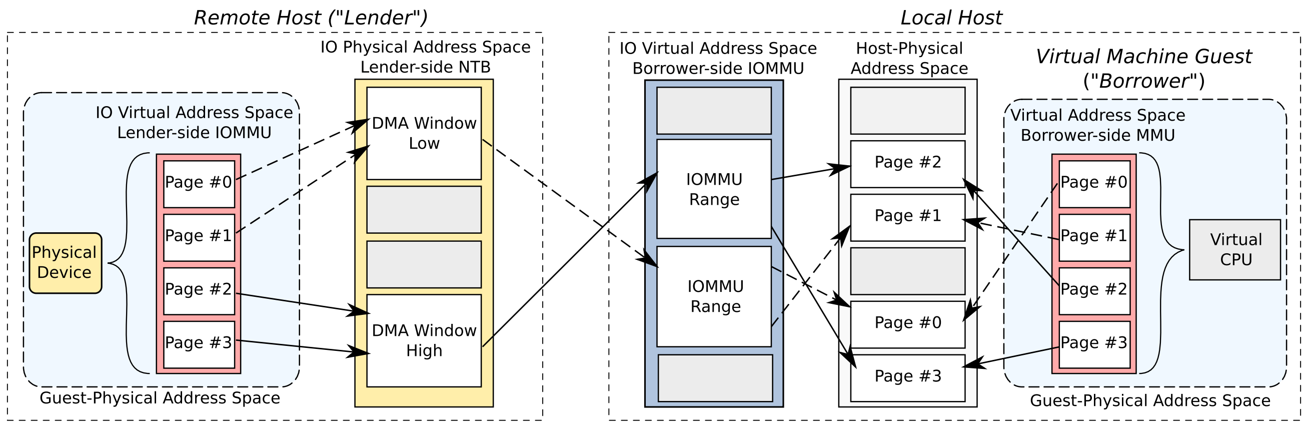


Fig. 7 By using IOMMUs on both sides of the NTB, it is possible to map a physically remote device into a local VM guest's address space. The borrower-side IOMMU provides continuous memory ranges that can be mapped over the NTB, while the lender-side IOMMU allows the device to be mapped into an address space using the same guest-physical addresses used by the VM

to know in advance what memory pages will be used until the VM instance is actually loaded and the guest OS boots up, because only then will the memory used by the VM actually be allocated. In addition, the mediated device API does not provide any information about the guest-physical memory layout, which we need to know which address ranges to map for the device.

However, in order for a device to do DMA, a dedicated register in the device's configuration space must be set. This register is common for all PCIe devices. Relying on the assumption that this register is disabled until the guest OS is booting up (and memory for the instance has been allocated), our solution intercepts when a configuration cycle enables this register, and only then notifies KVM to pin the necessary memory pages. With the pages now locked in memory, we are able to properly set up DMA windows to memory used by the VM instance. The x86 architecture uses well-defined addresses for low and high memory. We are able to discover how much memory the VM has allocated by attempting to pin memory starting at these addresses. In this way, we are able to dynamically detect the guest-physical memory layout.

Finally, VFIO and mediated devices use the *eventfd* API to trigger interrupts in the VM instance. Our current implementation intercepts calls to the configuration space that enables interrupts and sets up an interrupt handler on the lender-side. Whenever the device triggers an interrupt, the lender-side request handler is invoked. This handler must then notify the borrower, which in turn notifies the hypervisor using *eventfd*. This method is not ideal, as the latency of triggering an interrupt is increased. A benefit of our solution is that it allows us to enable legacy interrupts for devices borrowed by a VM, which is currently not supported when the borrower is a physical machine. We have also improved Device Lending in general with support for 64-bit MSI/MSI-X.

6 Supporting multiple devices and peer-to-peer

Some processing workloads may require the use of multiple I/O devices and/or compute accelerators, in addition to moving data between them in an efficient manner. This often involves the use device-to-device DMA, as described in Sect. 2.1, where a device is able to read from or write to the memory regions of other devices. However, as IOMMUs introduce a virtual address space for devices, TLPs must be routed through the root of the PCIe tree in order for the IOMMU to resolve virtual addresses. This means that shortest-route peer-to-peer transactions directly between devices in the fabric is not possible when using an IOMMU, and TLPs must traverse the root (Fig. 8). PCI-SIG has developed an extension to the transaction layer protocol that allows devices that have an understanding of I/O virtual addresses to cache resolved addresses [34], but this is not widely available as it requires hardware support in devices.

Because of this, the general perception among device vendors and driver developers has become that in order to make peer-to-peer transactions work efficiently, the

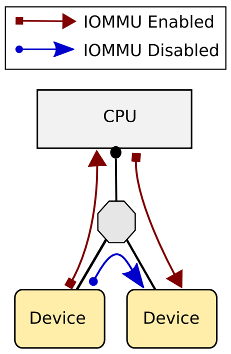


Fig. 8 IOMMUs introduce a virtual I/O address space for devices. Peer-to-peer transactions between devices is routed through the root in order for the IOMMU to resolve virtual addresses to physical addresses

IOMMU must be disabled. This has led to a situation where device drivers would indiscriminately use physical addresses when setting up peer-to-peer access between devices. For our original Device Lending implementation, this posed a challenge, as we rely on intercepting calls made by the device driver to inject our own mappings in order to make DMA across the NTB transparent. However, this changed with the 4.9 version of the Linux kernel, when the DMA API was extended with a unified method for setting up mappings between devices. This extension makes it possible for Device Lending to intercept when a device is mapping another device's memory regions.

However, as devices installed in different hosts reside in different address space domains, the local I/O address used by one host to reach a remote device is not the same address a different host would use to reach the same device. In order for a borrowed device, *source*, to reach another borrowed device, *target*, the borrower needs a mechanism to resolve virtual I/O addresses it uses to addresses that *source*'s lender would use to reach *target*. As such, our solution is as follows:

– If *target* is local to the borrower, setting up a mapping is trivial. The lender simply sets up DMA windows to the individual memory regions of *target*, similar to how it already has set up a DMA window to the borrower's RAM. The lender returns the local I/O addresses it would use to reach over the NTB to the memory regions of *target*. Note that this would work for any device in the borrower, not only those that are controlled by Device Lending.
– If *target* is locally installed in the same host as *source* (same lender), the lender simply sets up a local IOMMU mapping and returns the local I/O addresses to the memory regions of *target*. If IOMMU is disabled, then it is simply a matter of returning the local I/O addresses of memory regions of *target*.
– If *target* is a remote device (different lenders), the *source*'s lender creates DMA windows through the appropriate NTB to *target*'s lender. Note that this NTB may be different to the one used in order to reach the borrower. It then returns the memory addresses it would use to reach over the NTB to the memory regions of *target*.

The borrower, after receiving these lender-local I/O addresses, stores them along with its own virtual addresses to the memory regions of *target*. When the device driver using *source* calls the new DMA API functions to map the memory regions of *target* for *source*, we are able to look up the corresponding lender-local addresses and inject these. The driver can in turn initiate DMA, completely unaware of the location of both *source* and *target*, and the transfer will reach *target* through the correct NTB.

An additional consideration is required if the borrowing machine is a VM. In this case, *target* is already mapped into the guest-physical address space of the VM guest. The memory regions of *target* must be mapped for *source* using these exact addresses. Since the VM case already uses the lender-side IOMMU, as explained in Sect. 5, we can simply use the IOMMU of *source*'s lender and specify the addresses that correspond to the VM guest's view of the address space.

7 Performance evaluation

In order to evaluate our improved Device Lending implementation, we have done extensive evaluations of the bandwidth and latency of peer-to-peer DMA transfers. As VM pass-through require the use of an IOMMU on the lending system, we particularly focus on the impact I/O address virtualization has on performance with regards to longer data paths. For all our comparisons, we present the topology and machine configurations and compare performance for native bare-metal borrowers and VM borrowers. Our baseline comparison for all evaluations are running locally, on a bare-metal machine.

In Sect. 7.5, we prove the capability of running unmodified software and device drivers by presenting the performance of an unmodified convolutional neural network-based application, using the Keras framework with Tensorflow. We argue that running unmodified code using a complex machine learning framework on commodity hardware demonstrates the strength and flexibility of our Device Lending approach.

7.1 IOMMU performance penalty

Since IOMMUs create a virtual address space, TLPs need to be routed through the root of the PCIe tree in order to resolve virtual I/O addresses (Fig. 8). Processor designs are complex and often not well-documented, making it difficult to determine what exactly happens with the memory operations in progress once they leave the PCIe fabric and enter the CPUs. Memory operations may be buffered, awaiting IOMMU translations, or the IOMMU may need to perform a multi-level table look up for resolving addresses.

TLPs are either *posted* or *non-posted* operations, meaning that some transactions, such as memory reads, require a completion. Read requests are affected by the number of hops in the path between requester and completer; the longer the path, the higher the request-completion latency becomes. As the number of read requests in flight is limited by how many uncompleted transactions a requester is able to keep open, a longer path can potentially reduce performance. In addition, PCIe allows a completer to respond with less data at the time than is actually requested. For example, a read TLP requesting 256 bytes may terminate with 4 completions containing 64 bytes each, rather than a single completion with 256 bytes.

In order to isolate the consequence of TLPs being routed through the root, we have used the setup shown in Fig. 9. Two Intel Xeon machines are connected together with Dolphin's PXH830 NTB host adapters [35] and an external x8 PCIe cable. The lender has a PCIe switch on the motherboard, with both the NTB adapter and an Nvidia Quadro K420 GPU sitting below it. Note that since the K420 is Gen2 x16, we only need a Gen3 x8 link between the NTB adapters, as they provide approximately the same bandwidth.

For this evaluation, we have chosen to create a high-bandwidth workload using the *bandwidthTest* [36] program. This utility program is from the CUDA Toolkit samples. Choosing this program serves an additional purpose, demonstrating that Device Lending truly works with remote devices, without requiring changes to application or driver software. The bandwidth is measured running on the borrower, using the remote K420's onboard DMA engine to copy data between GPU memory and borrower's RAM. For each transfer size, *bandwidthTest* initiates 10 transfers and then report the mean bandwidth. We have repeated this 10 times.

Figure 10 shows the reported mean bandwidth for both DMA writes and DMA reads, comparing the performance of shortest path (Rem-SW) with TLPs being routed through the root (Rem-IOMMU). We observe that the reported bandwidth is reduced when the IOMMU is enabled, especially for the read performance. As mentioned, a PCIe completer is allowed to reply with multiple completions to a single request. In our case, using a PCIe tracer similar in concept to that of network packet tracers, we observe that the read TLPs are actually modified by the lender-side *CPUs* (and not the completer). The maximum TLP payload

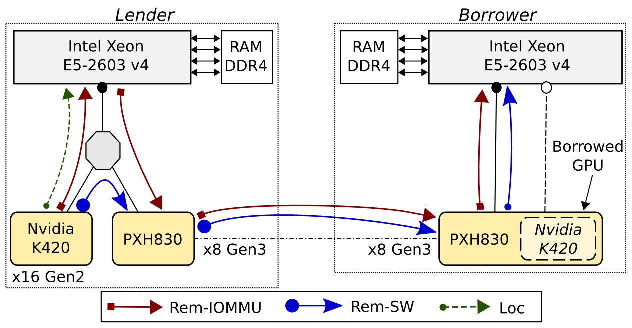


Fig. 9 Configuration used in our IOMMU evaluation. The borrower is using the remote GPU. When the lender-side IOMMU is enabled, TLPs are routed through the lender's root before going over the NTB (Rem-IOMMU). We have also compared to a baseline comparison, running locally on the lender machine itself (Loc)

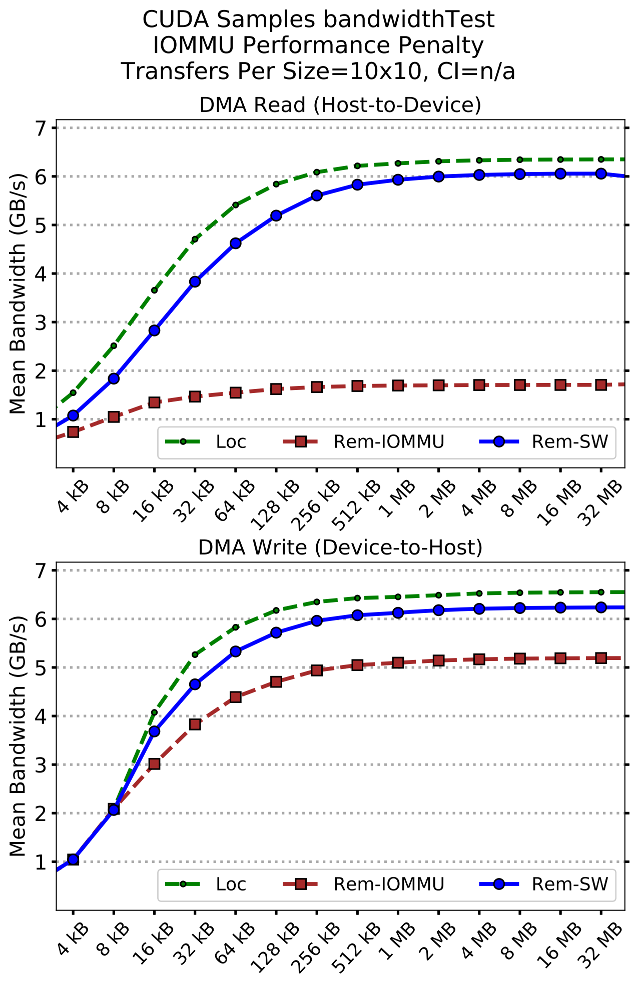


Fig. 10 Reported bandwidth for different transfer sizes using an unmodified version of the *bandwidthTest* CUDA samples program

size in our configuration is 256 bytes, meaning that devices can write or read up to 256 bytes per request. We observe, however, that every 256 byte request routed through the root is emitted out again as 4×64 byte read requests on the other side of the root. As read performance is already limited by the number of requests they are able to keep open, requesting less data at a time leads to very poor utilization of the link. Although not as bad as reads, write performance is also affected when the lender-side IOMMU is enabled.

Note that we also compared our results to running locally on the lender without using Device Lending (Loc). The achieved bandwidth of the local run is slightly better than our peer-to-peer performance for smaller transfer sizes; this is most likely due to the fact that the GPU is further away from the CPU running the driver, and therefore slightly increasing the time it takes to initiate a DMA transfer as well as other synchronization with the devices. We observe that for sizes of 1 megabyte and more, the significance of this additional latency decreases and the reported bandwidths starts to converge.

7.2 Native peer-to-peer evaluation

In order to evaluate our multi-device support, we have measured the performance of peer-to-peer DMA transfers between two Nvidia Quadro K420 GPUs. The machines are connected together using the PXH830 NTB adapters in a three-way configuration, providing a separate Gen3 x8 link between all three machines. The K420 GPUs are Gen2 x16, which provides roughly the same bandwidth as Gen3 x8.

Figure 11 shows the three different hardware configurations used in this evaluation:

- A local machine using two GPUs installed in the same local host, illustrated in Fig. 11a. This is our baseline for comparing the performance of using remote devices vs. local devices. Since it is not possible to enable peer-to-peer transfers on a local machine using the IOMMU, we instead force transfers to be routed through the root by placing the GPUs behind different PCIe switches.
- A local machine (borrower) using two remote GPUs, installed in a single remote host (one lender). This is illustrated in Fig. 11b.
- A local machine (borrower) using two remote GPUs, installed in different remote hosts (two lenders). This is illustrated in Fig. 11c.

A complete list of the scenarios are given in Table 1. Note that in Fig. 11, we have only highlighted the data path for peer-to-peer DMA writes with the IOMMU enabled and disabled. We compare the performance benefit of direct device-to-device DMA writes, using peer-to-peer transactions, to transfers via RAM, where one GPU first writes to RAM and the other reads from it using DMA. In order to do this, we have developed two CUDA [36] applications for measuring transfers from one GPU to another. Note that we do not use any special semantics or other userspace software to make this CUDA program work for borrowed remote GPUs, using Device Lending they simply appear to the CUDA programs as if they are locally installed.

7.2.1 Bare-metal bandwidth evaluation

The first of the two CUDA programs measures the bandwidth of DMA transfers from one GPU to another using two different transfer “modes”. The first mode is enabling peer-to-peer transactions, allowing one GPU to write directly into another GPUs memory. The second mode is hosting an intermediate buffer in system memory (RAM), where one GPU first writes to that buffer, followed by the other GPU reading from it afterwards. We record a CUDA event before and after each scheduled transfer, and we also schedule a dummy CUDA kernel launch in order to prevent our bandwidth measurements being affected by the CUDA driver’s ability to pipeline transfers.[3]

Figure 12 shows the bandwidth for all three configurations (depicted in Fig. 11). We have recorded the completion time for 1000 individual DMA transfers of a given size, for each transfer size shown along the X-axis, and plot the mean bandwidth. We also show the 95% confidence interval as a filled-out area around the respective lines. The top row shows our peer-to-peer transfers, while the bottom row shows transfers via system memory. We also show the difference in performance when the IOMMU is enabled and disabled on the lender(s), where the GPUs reside. Note that in our local comparison, we place the GPU behind a different PCIe switch in order to force TLPs to traverse the root, since it is not possible to enable the IOMMU in this scenario.

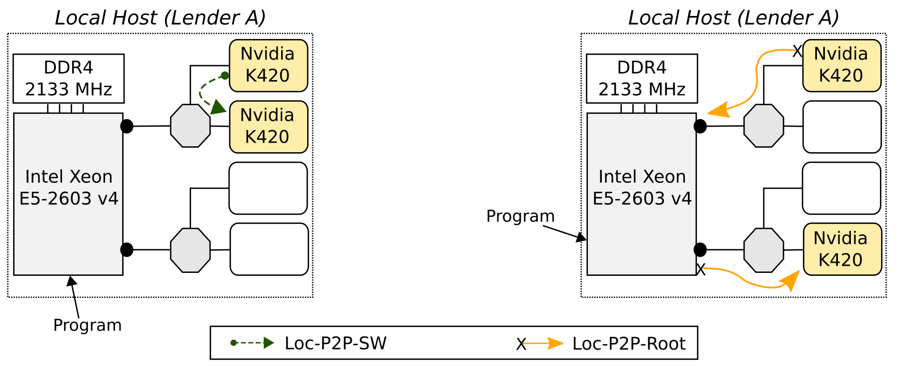
Using peer-to-peer DMA writes (top row), we see that the achieved bandwidth is almost the same as our local comparison in the same lender scenario: 1L-P2P-SW is almost identical to Loc-P2P-SW, and 1L-P2P-IOMMU is almost identical to Loc-P2P-Root. Even though the GPUs are remote, the data path between the GPUs are similar. For smaller transfer sizes, the local transfers achieve slightly higher bandwidth. However, when the transfer size increases, the lines converge, and for transfers of 4 megabyte and above, the difference becomes negligible. We suspect that the protocol used by the driver in order to synchronize the GPU and schedule DMA transfers may involve some reading and writing over the NTB. For small-sized transfers, this additional latency relative to the transfer size has an effect.

When the GPUs reside in different lenders, the data path is increased, which has an expected impact on performance. As shown in the 2L-P2P plot (top row, to the right) in Fig. 12, particularly when the IOMMU is enabled, the increased number of hops impacts the performance.

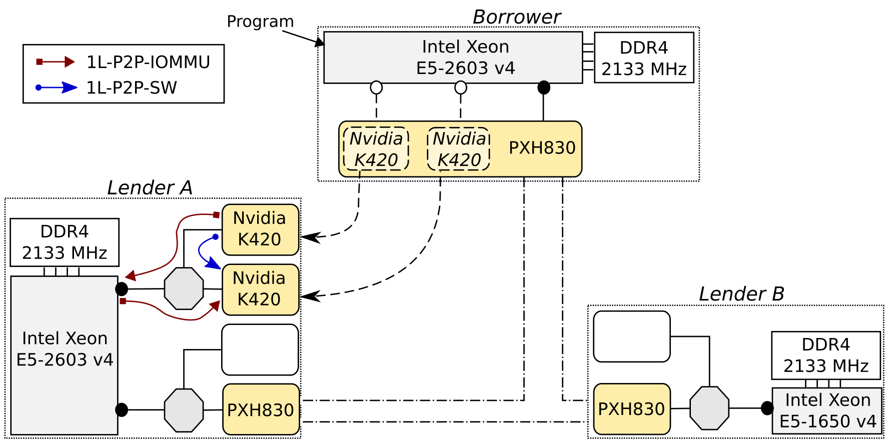
The second mode of our program was used to evaluate “bouncing” via system memory. By hosting a memory buffer in RAM, one GPU has to first write to this buffer before the other GPU in turn can read from it. Borrowing remote GPUs using Device Lending, the distance between system memory and GPU is now increased, and the impact of this is visible, as illustrated in Fig. 12 (bottom row). We see that transfers that do not cross the root (2L-RAM-SW)

[3] The CUDA samples *bandwidthTest* program, used in Sect. 7.1, schedules 10 rapid copy operations at the time and reports the average of these ten, allowing the CUDA driver to pipeline transfers and optimize small transfers.

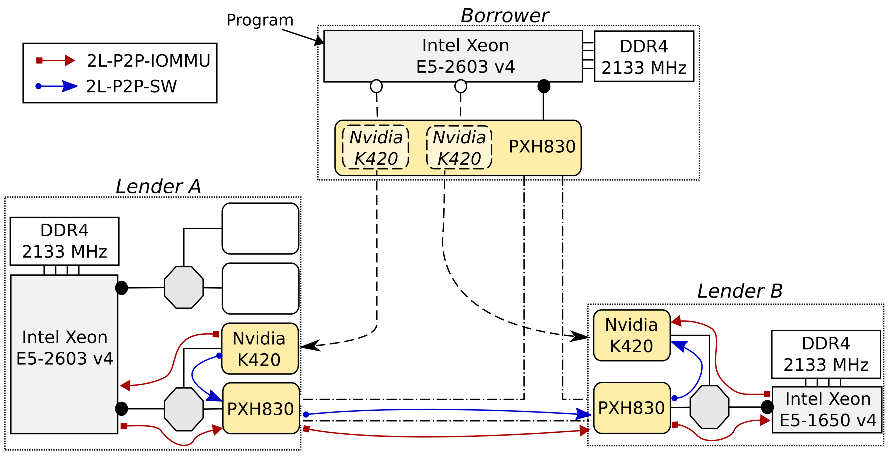
Fig. 11 The three-node cluster configurations used in our bare-metal multi-device evaluation, showing the the data paths for direct peer-to-peer write transactions



(a) Two GPUs used by a local instance. We force transactions to traverse the root in the Loc-P2P-Root scenario in lieu of an IOMMU, shown on the right-hand side.



(b) Two GPUs borrowed natively from the same lender. Note how the data paths are similar to the local scenarios in Figure 11a.



(c) Two GPUs borrowed natively from two different lenders.

are very similar to our baseline local comparison (Loc-RAM-Root). However, similarly to what we observed in Sect. 7.1, DMA reads are significantly affected by TLPs traversing the root, as this drastically reduces the link utilization. This is seen in 1L-RAM-IOMMU and 2L-RAM-IOMMU, where the reported bandwidth drops drastically.

Table 1 The different scenarios used in our bare-metal peer-to-peer evaluation. Note the number of hops and CPU roots transfers have to traverse

Name	Scenario	Transfer	Roots	Hops
Loc-P2P-SW	Two local GPUs installed in same machine as driver.	Peer-to-peer	0	1
Loc-P2P-Root	Two local GPUs installed in same machine as driver.	Peer-to-peer	1	3
Loc-RAM-Root	Two local GPUs installed in same machine as driver.	Via local RAM	1	3
1L-P2P-SW	Two remote GPUs borrowed from the same lender.	Peer-to-peer	0	1
1L-P2P-IOMMU	Two remote GPUs borrowed from the same lender.	Peer-to-peer	1	3
2L-P2P-SW	Two remote GPUs borrowed from different lenders.	Peer-to-peer	0	4
2L-P2P-IOMMU	Two remote GPUs borrowed from different lenders.	Peer-to-peer	2	8
1L-RAM-SW	Two remote GPUs borrowed from the same lender.	Via borrower's RAM	3	11
1L-RAM-IOMMU	Two remote GPUs borrowed from the same lender.	Via borrower's RAM	3	11
2L-RAM-SW	Two remote GPUs borrowed from different lenders.	Via borrower's RAM	1	8
2L-RAM-IOMMU	Two remote GPUs borrowed from different lenders.	Via borrower's RAM	3	11

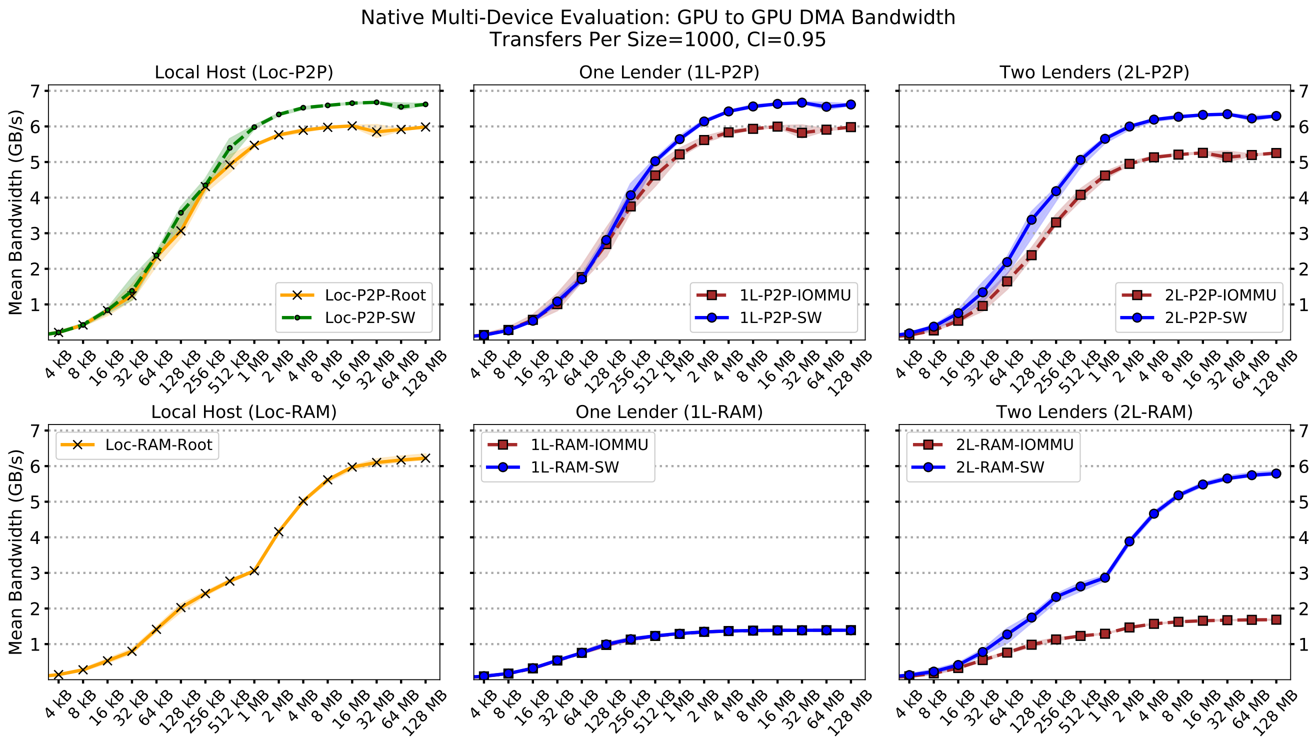


Fig. 12 Mean DMA bandwidth for different transfer sizes. The filled-out area represents the 95% confidence interval. The top row shows writes using peer-to-peer, while the bottom row shows "bouncing" via RAM. For the peer-to-peer, we achieve almost the same bandwidth as our local comparison. For transfers via RAM, the bandwidth is reduced by read TLPs traversing through CPU roots

It is interesting to note that 1L-RAM-IOMMU and 1L-RAM-SW both traverse the root, but the IOMMU is respectively on and off. This strengthens our suspicion that the issue is TLPs being routed through the root, and not necessarily some effect of using the IOMMU alone.

A simplified illustration of the data path for the full list of scenarios is shown in Fig. 13. Note that each additional "hop" in the path adds additional latency to the TLP completion time, something that particularly affects reads.

Our peer-to-peer bandwidth evaluation indicates that it is possible to achieve close to local performance. For DMA write operations, the performance of a local program using borrowed remote devices is comparable to using local devices. Note that while DMA reads are affected by the increased distance between the device and the memory it reads from, it is expected for longer data paths and not an effect of our Device Lending mechanism.

7.2.2 Bare-metal latency evaluation

Using CUDA, there are two ways of initiating DMA transfers; either the CPU can initiate DMA transfers, or the device can do it itself. The first approach is similar to the CUDA samples program *bandwidthTest*. The second approach is possible using CUDA's unified memory model, where a CUDA kernel can access system RAM directly through a memory pointer. This eliminates the need for an explicit copy to GPU memory operation. With unified memory, it is also possible for one GPU to directly access memory of another GPU, using peer-to-peer DMA.

As shown in Sect. 7.1, we suspect that the increased distance between CPU and device affects the time it takes for the driver to synchronize with the device and initiate a transfer. We also observed a similar effect for smaller-sized transfers in Fig. 12. Therefore, we developed a second CUDA program in order to measure peer-to-peer latency more accurately. Using CUDA kernels and allowing the GPUs themselves to initiate transfers, we eliminate any synchronization overhead caused by the driver (running on the local CPU). One GPU is tasked with increasing a counter, writing it to the other GPU's memory pointer and waiting for an acknowledgement before continuing (*ping*). The other GPU waits for the counter to increase by one, and acknowledges by writing back to the first GPU's memory pointer (*pong*). The whole roundtrip is measured by recording the current GPU clock cycle count and divide it by the clock frequency, giving us the full *ping–pong* latency.

The memory used for our counter can either be hosted in GPU memory or in RAM. The difference is that in the peer-to-peer scenarios we eliminate any DMA read operations and the GPUs are able to write directly to GPU memory. When memory is hosted in RAM, one GPU has to first write (over the NTB) to the borrower's RAM, and then the other GPU must read from the borrower's RAM (also over the NTB). The different data paths are illustrated in Fig. 13. Note that each additional "hop" in the total path adds additional latency to the overall completion time.

Figure 14 shows the mean ping-pong latency for all scenarios. We measured the latency for 100,000 ping-pongs, and the error bar depicts the 99% confidence interval. For comparison, the one-way RAM-to-RAM memory latency between the borrower and Lender B was measured to around 700 nanoseconds, where the NTB itself adds 350-365 nanoseconds. When GPUs reside behind the same switch (1L-P2P-SW), we achieve the same latency as our local comparison (Loc-P2P-SW). We also see the same when the IOMMU is enabled (1L-P2P-IOMMU) and the local comparison (Loc-P2P-Root). Again, this demonstrate that our Device Lending mechanism does not add any overhead.

We also observe that the latency increases according to the increased data paths (illustrated in Fig. 13), as expected. The latency for 2L-P2P-SW increase with a little more than 700 nanoseconds, compared to 1L-P2P-SW (and Loc-P2P-SW), which corresponds with the 350 nanoseconds added by the NTB (in one direction). In the scenarios where the counter memory is hosted in the borrower's RAM, the latency increases significantly because both GPUs now have to read in addition to writing. Our latency evaluation show that the latency of reading and writing is only affected by the path, and achieving the same latency as our local comparison when the path is similar.

7.3 VM peer-to-peer evaluation

We have also evaluated peer-to-peer performance for devices passed-through to a VM. We installed Ubuntu 16.04 with CUDA 9.0 on an Intel P4800X NVMe drive. As our VM emulator, we used QEMU 2.10.1. Two Nvidia

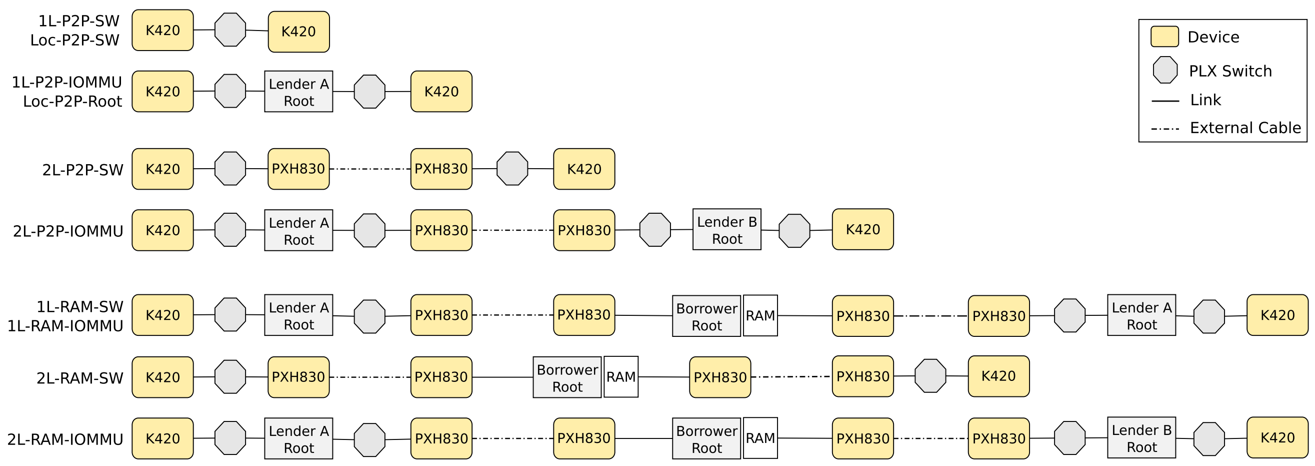


Fig. 13 Data paths for the different bare-metal scenarios. Each hop slightly increases the completion latency

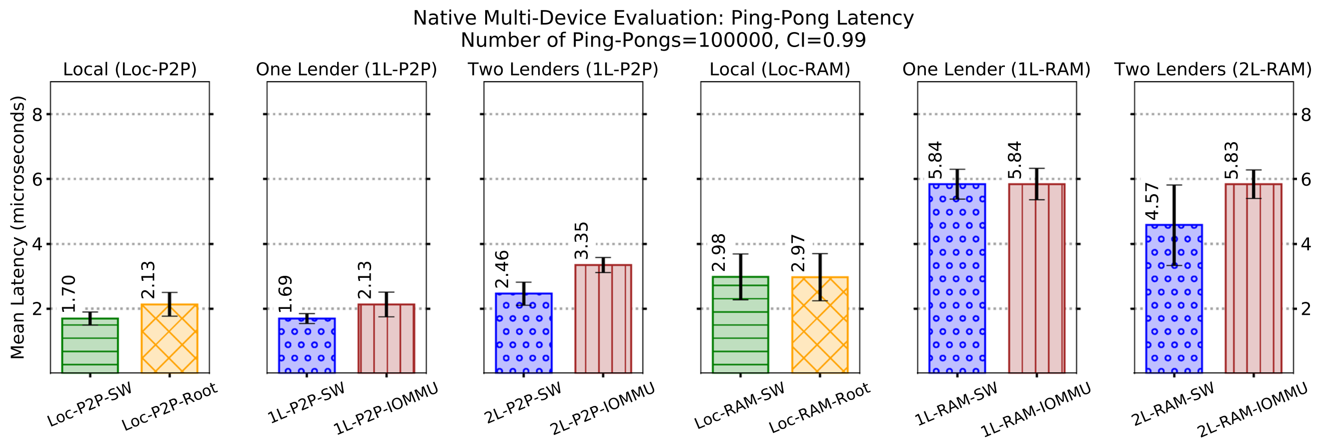


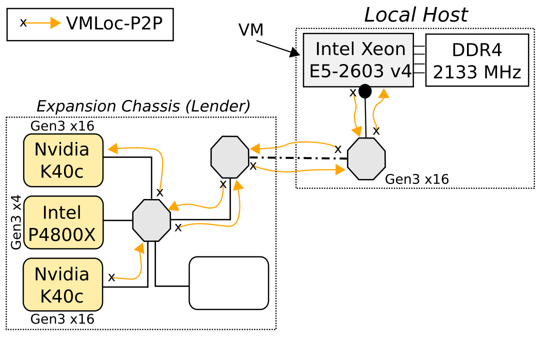
Fig. 14 Mean round-trip latency for 100,000 ping-pongs. The error bar represents the 99% confidence interval. For the peer-to-peer scenarios, we achieve the expected latency corresponding to the data path. When bouncing through RAM, the latency increases drastically due to the second GPU having to read from RAM

Tesla K40c GPUs along with the boot disk was passed through to a local VM using standard VFIO pass-through [32] with KVM, and to a remote VM using our Device Lending implementation. We used the same two CUDA programs from Sect. 7.2 for measuring bandwidth and latency respectively.

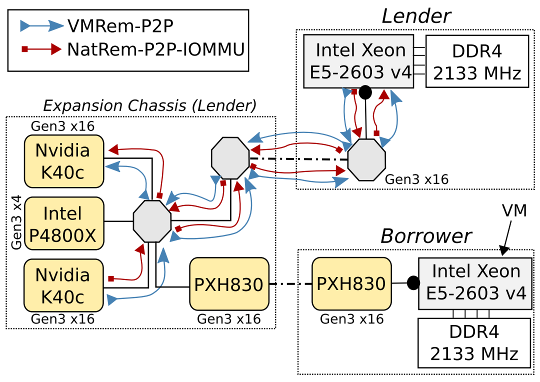
Figure 15 depicts the topologies used for these tests. The GPUs, the disk and the NTB adapter are located in an expansion chassis connected with a transparent link to the lender. TLPs must be routed through the lender's root before they can be transmitted over the NTB (which also resides in the expansion chassis), making this this configuration suboptimal for running a remote VM. As such, it serves as a worst-case scenario for running a VM, especially for the scenario where transfers are bounced via RAM. Figure 17 shows the data path for all scenarios. We have also included a native remote comparison using Device Lending where the IOMMU is enabled to illustrate any virtualization overhead. Note that the data path is similar for both local and remote scenarios when the devices use peer-to-peer DMA. The evaluated scenarios are listed in Table 2.

7.3.1 VM bandwidth evaluation

Figure 16 depicts the measured bandwidth for all configurations, using the same CUDA program as in Sect. 7.2. For each transfer size, we plot the mean reported bandwidth of 1000 transfers. We also show the 95% confidence interval as a filled-out area surrounding the plotted lines. The upper-most plot depicts direct peer-to-peer transfers between the GPUs. We compare our Device Lending mdev implementation, with two borrowed remote GPUs passed to a local VM (VMRem-P2P), to a local comparison, or baseline, where two local GPUs are passed to a local VM



(a) Local VM instance using VFIO pass-through.



(b) Remote VM borrower and remote native borrower. Note the data path is similar when the lender-side IOMMU is enabled.

Fig. 15 Topologies used in our VM peer-to-peer evaluation. We have compared a local VM using VFIO pass-through to a remote VM using our extended Device Lending. Note that the devices are located in an expansion chassis, which increases the number of hops to the lender

(VMLoc-P2P). As with our previous bandwidth evaluations, we see a similar pattern as before: timing and synchronization between driver and GPUs appear to affect smaller-sized transfer, but becomes less relevant when the

Table 2 The different scenarios used in our VM peer-to-peer evaluation. Since the GPUs and the NTBs are now attached in an expansion chassis, the number of hops is very high when the IOMMU is enabled

Name	Scenario	Transfer	Roots	Hops
VMLoc-P2P	Two GPUs installed in same, local expansion chassis.	Peer-to-peer	1	7
VMRem-P2P	Two GPUs installed in same, remote expansion chassis.	Peer-to-peer	1	7
NatRem-P2P-IOMMU	Two GPUs installed in same, remote expansion chassis.	Peer-to-peer	1	7
VMLoc-RAM	Two GPUs installed in same, local expansion chassis.	Via local VM's RAM	1	7
VMRem-RAM	Two GPUs installed in same, remote expansion chassis.	Via borrowing VM's RAM	3	21
NatRem-RAM-IOMMU	Two GPUs installed in same, remote expansion chassis.	Via borrower's RAM	3	21

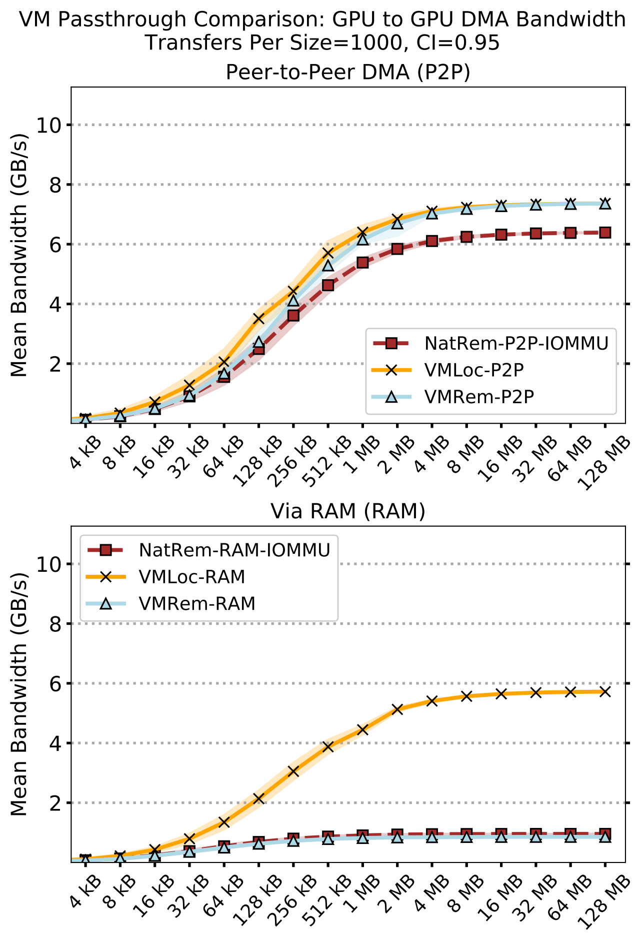


Fig. 16 Mean bandwidth for 1000 transfers per transfer size. 95% confidence interval. For peer-to-peer transactions we achieve the same bandwidth as running locally

transfer sizes increases. At around 4 megabytes this overhead is insignificant.

We have also included an additional comparison, namely a remote bare-metal machine borrowing the two remote GPUs and using them natively (NatRem-P2P-IOMMU). In order to force TLPs to traverse the same route as our KVM implementation (where lender-side IOMMU is required), the IOMMU is also enabled on the lender for the native comparison. It is interesting to note that it appears to achieve slightly lower bandwidth than when running in a VM, despite the data path being the same. We do not completely understand why this is the case.

The lower plot in Fig. 16 depicts transfers that are "bounced" via RAM. The memory buffer is allocated in system memory, and one GPU has to first write to it, before the other GPU can read from it. Since the lender's root is now even further away from the devices, read requests are significantly affected by the increased path. Combined with the reduced link utilization, as we observed in Sect. 7.1, the result is a drastic decrease in achieved bandwidth, for both our native Device Lending scenario (NatRem-RAM-IOMMU) and our KVM implementation (VMRem-RAM).

A simplified view of the data paths of all scenarios is illustrated in Fig. 17. We see that the path for NatRem-RAM-IOMMU and VMRem-RAM consists of 21 hops, traversing tree CPU roots, the NTB twice and the external transparent link four times. Note, however, that the performance for our VM implementation is similar to the native bare-metal performance, indicating that the Device Lending mechanism itself does not add any additional overhead. For the direct peer-to-peer DMA writes, the performance is comparable to the local comparison, which serves as our baseline.

7.3.2 VM latency evaluation

Using the second CUDA program, we also measured the ping-pong latency for the same scenarios. This is shown in Fig. 18, each bar is the mean reported latency for 100,000 ping-pongs (the error bar represents the 99% confidence interval). It is interesting to note that the latency for the remote scenarios using Device Lending (NatRem-P2P-IOMMU and VMRem-P2P) is actually slightly better than our local comparison, even though the data path is the same (Fig. 17). We assume this may be related to how VFIO exposes the GPU registers to the driver in the local case. In our KVM implementation, we expose the device memory regions directly, allowing the driver running in the VM guest to access GPU registers directly.

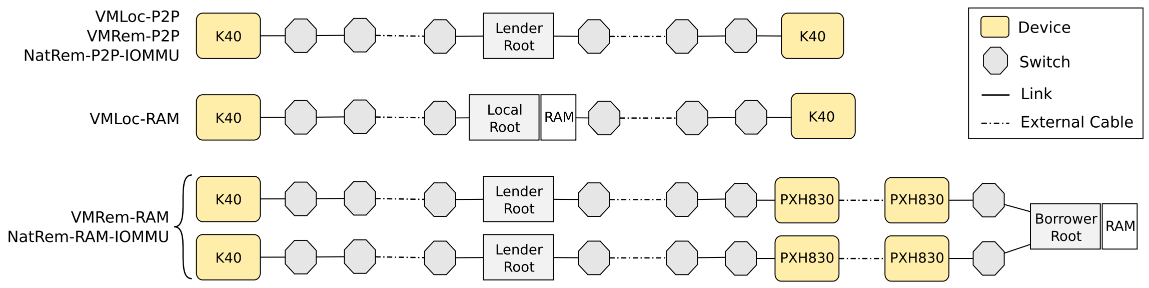


Fig. 17 Data paths for the different VM scenarios. Each hop slightly increases the completion latency. Because the NTB adapter is in the expansion chassis next to the GPUs, the number of hops when the lender-side IOMMU is enabled is very high

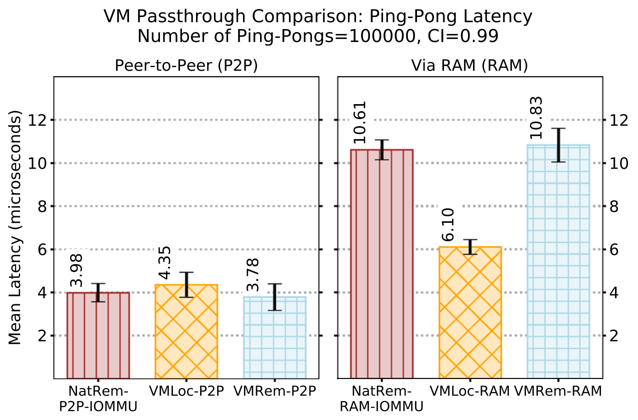


Fig. 18 Mean round-trip latency for 100,000 ping-pongs. 99% confidence interval. For peer-to-peer transactions, we achieve the same latency as running locally

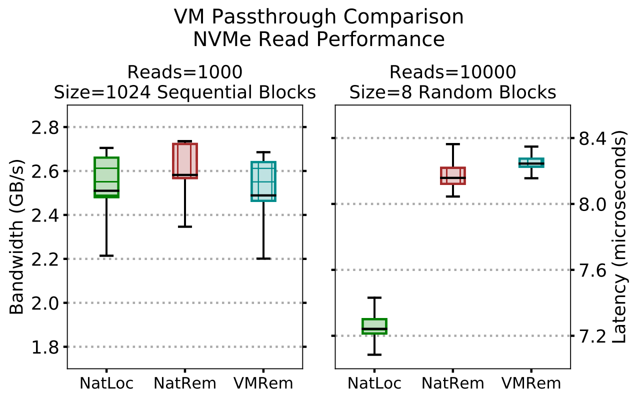


Fig. 19 Bandwidth and latency when reading from disk (DMA write). We read 1024 sequential blocks for measuring bandwidth, and 8 blocks with a random offset for latency

The increased latency of reading from remote RAM corresponds with the increased number of hops. The data path of running the VM locally (VMLoc-RAM) has only 7 hops, while the data paths of our remote native comparison (NatRem-RAM-IOMMU) and the remote VM (VMRem-RAM) both have 21 hops.

Our VM peer-to-peer evaluation indicate that we are able to achieve the same performance as a local VM when the data path is the same, and that we achieve the same performance as running natively (with lender-side IOMMU) even in the worst-case scenario.

7.4 Pass-through NVMe experiments

We have also performed experiments with our VM implementation using an Intel Optane 900P NVMe disk. We have compared the performance of the disk on a local machine without using Device Lending, a physical borrower (NatRem), and from a VM guest (VMRem). The machines are connected back-to-back using PXH830 NTB adapters [35]. The one-way RAM-to-RAM latency was measured to 550–580 nanoseconds, where the NTB adds around 350–370 nanoseconds. We have used QEMU 2.10.1 as our VM emulator, and running Ubuntu 16.04 as the guest OS. Note that while any guest OS would be possible, including Microsoft Windows, we have chosen Linux in order to run the same benchmarking code on a physical borrower, as well as locally on the lender. Device Lending requires Linux on the host.

Figure 19 shows the bandwidth for reading 1024 sequential blocks repeated 1000 times. One block is 512 bytes. There is very little difference in the achieved bandwidth, except for a few additional outliers for our VM borrower (VMRem). Interestingly, we observe that the physical borrower (NatRem) achieves slightly higher median bandwidth than compared to the local baseline.

Latency was measured by reading 8 blocks repeated 10,000 times, each time at a random offset. Here, we observe that the difference between running locally and on the physical borrower is an increase of a little less than 1 microsecond. As the device now sits remotely, it has to first reach over the NTB once in order to retrieve the I/O commands, and then reach over the NTB again in order to post the I/O completion. This adds 700-730 nanoseconds to the latency, and is therefore an expected increase. We observe that passing the disk to a VM running on the borrower (VMRem), only increases the latency slightly compared to the physical borrower (NatRem). Our evaluation show that it is possible to borrow a remote NVMe drive without any performance overhead beyond the added latency of the NTB. Additionally, it shows that our KVM extension to Device Lending is able to achieve almost the same bandwidth and latency as a native borrower.

7.5 Image classification workload

In order to demonstrate that Device Lending is applicable for real-world workloads, we run a GPU-intensive machine learning task. The program we use for the tests is a typical implementation of convolutional neural network (CNN) training in the Python machine learning framework Keras [37]. Keras is a higher level framework and wraps different lower level machine learning frameworks. In our case, Keras uses Tensorflow [38] as its backend. Keras allows multiple GPUs to work together by replicating the machine learning model being trained on each of the GPUs, then splitting the model's inputs into sub-batches which are distributed on the GPUs. When the GPUs are done, the sub-batches are concatenated on the CPU into one batch. This introduces quasi-linear speedup [39]. However, as our machine learning program can only run on a single system, we utilize multi-GPU support in Keras by borrowing remote GPUs and making them appear locally installed using Device Lending.

Our real-world workload is produced by a program that trains available models in Keras on given datasets with given hyperparameters using transfer learning [40]. Transfer learning is a technique for training datasets, where we use models with weights that are pre-trained on a dataset with similar classes to the dataset we want to train on. In our case, we use a VGG19 [41] model pre-trained on the ImageNet [42] dataset.

Transfer learning is done in two training steps: first, we remove the classification block of the pre-trained model, attach a new block corresponding to the number of classes in the dataset we want to train on, and train the new classification block only. In the second step, we train all the layers. For both steps, we use the stochastic gradient descent optimizer available in Keras. We ran the training on an 8-classes image dataset of the gastrointestinal tract called Kvasir [43–45].

We measured the runtime of a single epoch of the model training on two Nvidia K40c GPUs as well as loading images from storage and writing the results back using an Intel Optane P4800X. While a single epoch may not give the statistical significance needed for reliable machine learning results, we are only interested in the system performance. We used Keras 2.2.4 with a Tensorflow backend running on an Ubuntu 16.04 installation with CUDA 9.0 cuDNN 7.1. Both GPUs and the disk were used in all scenarios, and we also booted VMs and physical machines from the disk. For the native remote tests (NatRem-SW and

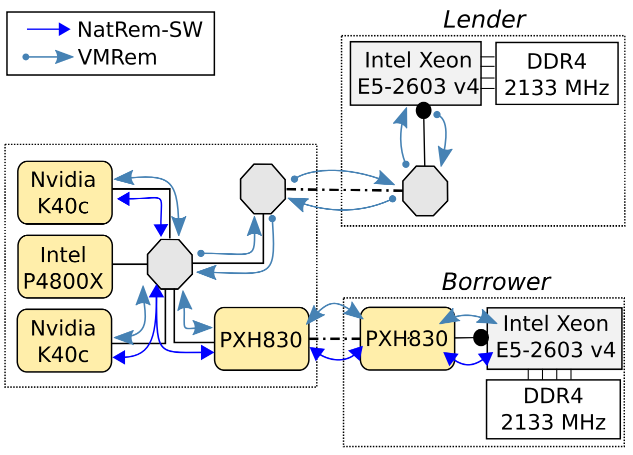


Fig. 20 Configuration used for our workload. The we have run a local native (NatLoc) and local VM (VMLoc) comparison on the lender and remote runs on the borrower (NatRem and VMRem). Note that this topology best-case scenario for the remote *native* peer-to-peer data path (NatRem-SW), while simultaneously being worst-case for remote VMs (VMRem)

NatRem-IOMMU) the disk was instead locally installed, in order to boot from it. The physical host had 16 GB memory and 6 CPU cores (Intel Xeon CPU E5-2603 v4). We reserved 4 cores and 8 GB for the VM, and used all 6 cores and the remaining 8 GB for the native run.

Figure 20 depicts the topology of our evaluation. When using the multi-GPU model in Keras, the Tensorflow backend outputs a GPU peer-to-peer matrix, indicating that it is capable of direct DMA without bouncing via RAM.[4] We have used the same configuration for local and remote runs. As discussed in Sect. 7.3, this topology is a form of worst-case scenario for running remote VMs because the IOMMU address virtualization requires TLPs to be routed through the lender's root. At the same time, it is a best-case scenario for native runs with direct peer-to-peer transfers, as devices reside behind the same switch.

Figure 21 show the total runtime of the model training for the different scenarios. For the best-case scenario, running natively with direct data paths, we see that the remote run (NatRem-SW) runs as fast as the local native comparison (NatLoc). Enabling the IOMMU and forcing traffic through the lender's root (NatRem-IOMMU) increases the overall runtime. We have also compared a local VM using standard VFIO pass-through (VMLoc) to our KVM implementation (VMRem). It is interesting to note that the local VM runtime is higher than the remote native using the IOMMU. This indicates that virtualization adds some additional overhead compared to running on bare-metal. The VMs also use less memory and CPU cores than native. We suspect this is also the case for the remote

[4] We also confirmed that the GPU driver sets up peer-to-peer transfers by observing IOMMU mappings ranges.

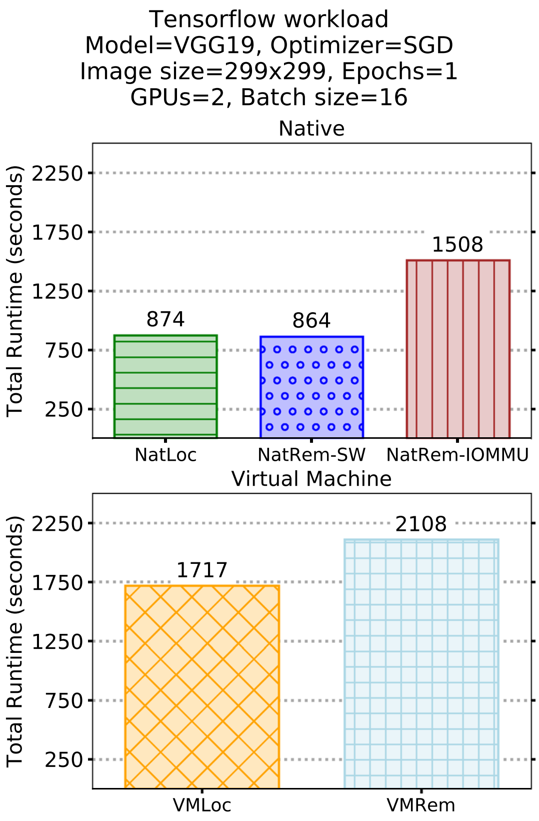


Fig. 21 Total runtime of Keras workload in different scenarios. There is a significant performance decrease when running in a VM and when the GPUs are remote. Note that the local VM (VMLoc) performs worse than the native remote, indicating that there is additional performance overhead caused by virtualization

VM (VMRem), which means that we get virtualization overhead in addition to the performance penalty of longer data paths through the IOMMU.

8 Discussion

Device Lending is a mechanism for decoupling devices from the hosts they physically reside in. Using hardware memory mappings, we facilitate the use of remote hardware resources without adding any software overhead [1, 2, 45]. We have extended our original Device Lending with KVM support for peer-to-peer transfers between multiple devices passed through to a VM. In this section, we discuss some considerations for borrowing devices from a VM guests.

8.1 I/O address virtualization

In our performance evaluation (Sect. 7), we observed that the data path in terms of number of hops affects the TLP completion latency. We also observed that using the lender-side IOMMU forces TLPs to be routed through the CPU on the lender. Our findings also seem to match previous performance evaluations of IOMMUs [46].

When the driver and the device frequently communicate with each other, as seen as synchronization overhead for small DMA transfers in our evaluations using Nvidia GPUs, it may affect performance since TLPs has to go back and forth over NTB. For larger DMA transfers, we observed that the significance of this delay decreases. For peer-to-peer transfers that do not require synchronization by the CPU, as is the case for our ping-pong evaluations, the distance between GPU and driver is insignificant. It should be noted that traversing the NTB adds less than half of the latency added by InfiniBand FDR adapters [15, 21]. For native peer-to-peer transfers with PCIe switches, where shortest-path routing is possible, we therefore argue that Device Lending can be used with extremely low performance overhead.

A major performance bottleneck occurs when DMA read requests are routed through the root, as the Intel Xeon CPUs used in our evaluations alter the read requests to request less data at the time (from 256 to 64 bytes). This leads to decreased utilization of the PCIe links. Since devices may be limited by the number of read requests they are able to keep open, the combination of poor link utilization and longer data paths can drastically affect the DMA bandwidth for some scenarios. However, we also observed a similar effect when read requests were routed through the CPU without the IOMMU being enabled. This strongly indicates that routing peer-to-peer through the CPU is a problem in general. Since the I/O address virtualization is required for *both* local and remote pass-through, it is worth investigating further by evaluating other CPU architectures that implement an IOMMU, such as AMD EPYC/Zen and IBM Power.

Our recommendation is to try to minimize the number of hops after the CPU in order to reduce the performance penalty of routing through the root, and to use shortest-path peer-to-peer transfers where possible. For bare-metal borrowers, this can be accomplished by disabling the IOMMU on the lender all together. For VMs, it may be possible to create a PCIe backplane that uses an NTB per device, allowing the NTBs to map the guest-physical address space for the devices rather than using an IOMMU for this.

Another possibility for avoiding IOMMU performance penalty, is using PCIe switches and devices that support caching of resolved virtual addresses using the standard for this specified by PCI-SIG [34]. Note that while it is also possible to disable the IOMMU on the borrower as well, this requires mapping the entire address space of borrower through the lender-side NTB and is therefore not practical with multiple borrowers. It also has little impact on peer-to-peer performance, unless one of the peering devices are local.

8.2 VM migration

Since Device Lending decouples devices from their physical location, our KVM implementation makes it possible to shutdown, migrate and restart a VM on a different host in the cluster (cold migration). The guest will retain access to the same physical devices. We demonstrated this in VM evaluation (Sect. 7.3) and in our image classification workload (Sect. 7.5), where the OS image with all the installed software and device drivers resides on the same boot disk that is being used by the remote and local VM guests, the native remote host, and the native local host comparison.

With proper emulator support, it would also be possible to hot-add and hot-remove devices to a running VM instance. Using such hot-swap functionality, migrating a VM while it is running could be achieved by first removing all devices before migrating and then re-attaching them afterwards. However, this would temporarily disrupt their use and force guest drivers to reset all devices.

A strong candidate for future improvements is looking into real hot-migration techniques, remapping devices while they are in use and without (or with minimal) disruption. However, such a solution would be non-trivial. Not only would it require keeping memory consistent during the migration warm-up, but DMA TLPs could potentially be in-flight during the migration. A mechanism for rerouting TLPs without violating the strict ordering required by PCIe must be implemented, which most likely will require hardware-level support.

8.3 Security considerations

A VM may allocate several GB of memory, which may be scattered in physical memory. In order to conserve mapping resources, we use the IOMMU on the local system in order to provide linear continuous memory ranges that are trivially mapped over the NTB. However, pass-through uses the IOMMU in order to match I/O addresses with the guest physical memory layout. Furthermore, VFIO requires that passed-through devices are placed in an IOMMU domain per VM, in order to provide isolation. In our case, this is not possible since we already use the IOMMU, and the virtual device is in another domain.

However, we use the IOMMU on the *lender* instead to map I/O virtual addresses to guest physical memory layout and provide the necessary memory isolation. This guarantees that the device is only able to access the specific DMA windows to the VM it is assigned to, and the IOMMU on the borrower guarantees that the same windows can only be used to access the VM memory. Our solution therefore provides the same level of memory isolation as standard pass-through. It is also not possible for software running in the VM to access memory outside the device memory regions of assigned devices.

8.4 Interrupt forwarding

For VMs, we register an interrupt handler on the device-side lender and forward interrupts to the local borrowing system, as explained in Sect. 5. A benefit with this approach is that we are able to support all types of PCIe interrupts, legacy, MSI and MSI-X, while native Device Lending only supports MSI and MSI-X. However, this introduces additional latency and involves software handling on the lender.

An evaluation is needed to determine what impact increased latency for interrupts may have on the performance of device drivers. As this impact most likely is not negligible, a candidate for improvement is therefore to use the same approach as bare-metal Device Lending for MSI and MSI-X, and map these types of interrupts over the NTB. This would remove any special software handling other than on the borrowing system alone, where we still need to use the *eventfd* API in order to notify the VM.

9 Conclusion

In this paper, we presented how we have extended our Device Lending implementation with support for the KVM hypervisor, allowing pass-through of physically remote devices to local VM guests. By dynamically probing the available memory and fully supporting both MSI and MSI-X interrupts, we have greatly improved the usability of our previous Device Lending implementation [2]. With dynamic memory layout detection, it is possible to compose custom configurations of distributed I/O resources in a PCIe cluster, for both native and virtual machines. Our experimental evaluations prove that we are able to compose flexible configurations of remote devices and enable dynamic time-sharing of resources using Device Lending. Being able to scale by dynamically reassigning devices to machines that currently need them, makes it possible to support a flexible I/O infrastructure that meet processing requirements and at the same time makes it possible to optimize resource utilization.

We have also implemented support for borrowing multiple devices from different lenders and enabled peer-to-peer access between them, allowing remote I/O resources to be used as if they were attached to the same local fabric. This allows physically remote devices to be used by the local system, without requiring *any* modifications to either device drivers or applications and without adding *any* software overhead in the data path. As part of this evaluation, we have investigated the impact of I/O address virtualization on performance. Specifically, we have performed bandwidth and latency measurements for different data paths. By enabling peer-to-peer transfers and routing shortest path between devices, we demonstrate that native Device Lending does not add a performance overhead in the data path beyond what is expected for longer paths. However, our results indicate that a major performance bottleneck occurs when DMA read requests traverse the CPU root, as is the case when the IOMMU on the lender is enabled. The Intel Xeon CPUs used in our evaluation alters the requests in a way that leads to poor link utilization. This impacts our VM implementation, as it requires the use of device-side IOMMU in order to map the device to guest-physical address space. This warrants further evaluations of other CPU architectures.

We have also run a real-world medical imaging classification application with borrowed remote hardware resources. We compare a best-case bare-metal topology for local and remote devices, and show that we achieve close to local performance using Device Lending. We have also compared our newly implemented VM support to a local VM, and show that it is possible to run such a workload in a VM using remote physical devices. We argue that being able to run the exact same code using remote GPUs and hard disks as if they were locally installed, thus making use of a complex machine learning framework with one of the most complex GPU implementations on the market, demonstrate the strength of Device Lending.

Acknowledgements This work has been performed mainly in the context of the BIA project *PCIe* (#235530) funded by the Research Council of Norway, with contributions from the *LADIO* project (EU Horizon 2020 #731970). The authors would like to thank Stig Magnus Baugstø, Halvor Kielland-Gyrud, Roy Nordstrøm and Hugo Kohmann at Dolphin Interconnect Solutions. We also thank Kristoffer Robin Stokke for feedback on the manuscript.

References

1. Kristiansen, L.B., Markussen, J., Stensland, H.K., Riegler, M., Kohmann, H., Seifert, F., Nordstrøm, R., Griwodz, C., Halvorsen, P.: Device Lending in PCI Express Networks. In: Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV, pp. 10:1–10:6 (2016). https://doi.org/10.1145/2910642.2910650
2. Markussen, J., Kristiansen, L.B., Stensland, H.K., Seifert, F., Griwodz, C., Halvorsen, P.: Flexible device sharing in pcie clusters using device lending. In: Proceedings of the International Conference on Parallel Processing Companion, ICPP Companion, pp. 48:1–48:10 (2018). https://doi.org/10.1145/3229710.3229759
3. Fountain, T., McCarthy, A., Peng, F.: PCI express: an overview of PCI express, cabled PCI express and PXI express. In: Proceedings of International Conference on Accelerator & Large Experimental Physics Control Systems, ICALEPCS (2005)
4. Peripheral Component Interconnect Special Interest Group (PCI-SIG): PCI Express 3.1 Base Specification (2010). https://pcisig.com/specifications
5. Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., Weigert, J.: Intel virtualization technology for directed I/O. Intel Technol. J. **10**(03) (2006) https://doi.org/10.1535/itj.1003.02
6. Linux IOMMU Support. https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt
7. Nvidia Virtual GPU Technology (vGPU). http://www.nvidia.com/object/virtual-gpus.html
8. Peripheral Component Interconnect Special Interest Group (PCI-SIG): Single-root I/O Virtualization and Sharing Specification (2010). https://www.pcisig.com/specifications/iov/single-root/
9. Ravindran, M.: Extending Cabled PCI Express to Connect Devices with Independent PCI Domains. In: Proceedings of the IEEE Systems Conference, SysCon, pp. 1–7 (2008). https://doi.org/10.1109/SYSTEMS.2008.4519048
10. Regula, J.: Using Non-transparent Bridging in PCI Express Systems. PLX Technology Inc, Sunnyvale (2004)
11. Sullivan, M.J.: Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge. Specification, Intel Corporation (2010)
12. Saito, K., Anai, K., Igarashi, K., Nishikawa, T., Himeno, R., Yoguchi, K.: ATM bus system (1998)
13. Alnæs, K., Kristiansen, E.H., Gustavson, D.B., James, D.V.: Scalable coherent interface. In: Proceedings of International Conference on Computer Systems and Software Engineering, CompEuro, pp. 446–453 (1990). https://doi.org/10.1109/CMPEUR.1990.113656
14. The Case Against iWARP (2015). https://www.chelsio.com/wp-content/uploads/resources/iWARP-Myths.pdf
15. RoCE vs. iWARP Competitive Analysis (2017). http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf
16. Trivedi, A., Metzler, B., Stuedi, P.: A case for RDMA in clouds. In: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys, pp. 17:1–17:5 (2011). https://doi.org/10.1145/2103799.2103820
17. Huang, J., Ouyang, X., Jose, J., Wasi-Ur-Rahman, M., Wang, H., Luo, M., Subramoni, H., Murthy, C., Panda, D.K.: High-performance design of hbase with RDMA over InfiniBand. In: Proceedings of International Parallel and Distributed Processing Symposium, IPDPS, pp. 774–785 (2012). https://doi.org/10.1109/IPDPS.2012.74
18. Jiang, W., Liu, J., Jin, H.W., Panda, D.K., Gropp, W., Thakur, R.: High performance MPI-2 one-sided communication over InfiniBand. In: Proceedings of International Symposium on Cluster Computing and the Grid, CCGrid, pp. 531–538 (2004). https://doi.org/10.1109/CCGrid.2004.1336648
19. Duato, J., Pena, A., Silla, F., Mayo, R., Quintana-Ortí, E.: rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In: Proceedings of International Conference on High Performance Computing and Simulation, HPCS pp. 224–231 (2010). https://doi.org/10.1109/HPCS.2010.5547126
20. Venkatesh, A., Subramoni, H., Hamidouche, K., Panda, D.K.: A high performance broadcast design with hardware multicast and GPUDirect RDMA for streaming applications on Infiniband clusters. In: Proceedings of International Conference on High

Performance Computing, HiPC (2014). https://doi.org/10.1109/HiPC.2014.7116875
21. Rosetti, D.: Benchmarking GPUDirect RDMA on Modern Server Platforms (2014). http://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/
22. Daglis, A., Novaković, S., Bugnion, E., Falsafi, B., Grot, B.: Manycore network interfaces for in-memory rack-scale computing. ACM SIGARCH Comput. Archit. News **43**(3), 567–579 (2015). https://doi.org/10.1145/2872887.2750415
23. Costa, P., Ballani, H., Razavi, K., Kash, I.: R2c2: a network stack for rack-scale computers. ACM SIGCOMM Comput. Commun. Rev. **45**(4), 551–564 (2015). https://doi.org/10.1145/2829988.2787492
24. Whitby-Strevens, C.: The transputer. ACM SIGARCH Comput. Archit. News **13**(3), 292–300 (1985). https://doi.org/10.1145/327070.327269
25. Hayes, J.P., Mudge, T., Stout, Q.F., Colley, S., Palmer, J.: A microprocessor-based hypercube supercomputer. IEEE Micro **6**(5), 6–17 (1986). https://doi.org/10.1109/MM.1986.304707
26. Peripheral Component Interconnect Special Interest Group (PCI-SIG): Multi-root I/O Virtualization and Sharing Specification (2008). https://www.pcisig.com/specifications/iov/multi-root/
27. Suzuki, J., Hidaka, Y., Higuchi, J., Baba, T., Kami, N., Yoshikawa, T.: Multi-root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device. In: Proceedings of Symposium on High Performance Interconnects, HOTI, pp. 25–31 (2010). https://doi.org/10.1109/HOTI.2010.21
28. Tu, C.C., Lee, Ct, Chiueh, Tc: Secure I/O device sharing among virtual machines on multiple hosts. ACM SIGARCH Comput. Archit. News **41**(3), 108–119 (2013). https://doi.org/10.1145/2508148.2485932
29. Tu, C.C., Chiueh, T.c.: Seamless fail-over for PCIe switched networks. In: Proceedings of the International Systems and Storage Conference, SYSTOR, pp. 101–111 (2018). https://doi.org/10.1145/3211890.3211895
30. Dilk, P.: Microsemi Switchtec PAX: Advanced fabric gen3 pcie switch (2017). https://www.youtube.com/watch?v=OB7OuektR0E
31. Wong, H.: PCI express multi-root switch reconfiguration during system operation (2011)
32. VFIO—“Virtual Function I/O”. https://www.kernel.org/doc/Documentation/vfio.txt
33. Jia, N., Wankhede, K.: VFIO Mediated Devices. https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt
34. Peripheral Component Interconnect Special Interest Group (PCI-SIG): Address Translation Services Revision 1.1 (2009). https://www.pcisig.com/specifications/iov/ats/
35. PXH830 Gen3 PCI Express NTB Host Adapter. http://www.dolphinics.no/products/PXH830.html
36. CUDA Toolkit Documentation v10.1.105 (2019). http://docs.nvidia.com/cuda/
37. Keras (2015). https://keras.io
38. TensorFlow: Large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org/
39. Keras documentation: multi_gpu_model (2015). https://keras.io/utils/#multi_gpu_model
40. Borgli, R., Halvorsen, P., Riegler, M., Stensland, H.K.: Automatic hyperparameter optimization in keras for the mediaeval 2018 medico multimedia task. In: Working Notes Proceedings of the MediaEval 2018 Workshop (2018)
41. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR arXiv (2014). arXiv:abs/1409.1556
42. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database. In: Proceedings of the Conference on Computer Vision and Pattern Recognition, CVPR (2009). https://doi.org/10.1109/CVPR.2009.5206848
43. Pogorelov, K., Randel, K.R., Griwodz, C., Eskeland, S.L., de Lange, T., Johansen, D., Spampinato, C., Dang-Nguyen, D.T., Lux, M., Schmidt, P.T., Riegler, M., Halvorsen, P.: KVASIR: A multi-class image dataset for computer aided gastrointestinal disease detection. In: Proceedings of the ACM Multimedia Systems Conference, MMSys, pp. 164–169 (2017). https://doi.org/10.1145/3083187.3083212
44. Hicks, S.A., Riegler, M., Pogorelov, K., Ånonsen, K.V., de Lange, T., Johansen, D., Jeppsson, M., Randel, K.R., Eskeland, S., Halvorsen, P.: Dissecting deep neural networks for better medical image classification and classification understanding. In: Proceedings of International Symposium on Computer-Based Medical Systems, CBMS (2018). https://doi.org/10.1109/CBMS.2018.00070
45. Pogorelov, K., Ostroukhova, O., Jeppsson, M., Espeland, H., Griwodz, C., de Lange, T., Johansen, D., Riegler, M., Halvorsen, P.: Deep learning and hand-crafted feature based approaches for polyp detection in medical videos. In: Proceedings of International Symposium on Computer-Based Medical Systems, CBMS, pp. 381–386 (2018). https://doi.org/10.1109/CBMS.2018.00073
46. Neugebauer, R., Antichi, G., Zazo, J.F., Audzevich, Y., López-Buedo, S., Moore, A.W.: Understanding PCIe performance for end host networking. In: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM, pp. 327–341 (2018). https://doi.org/10.1145/3230543.3230560

Jonas Markussen Jonas Markussen is a PhD student at Simula Research Laboratory, where his research is focused on new ways to use Non-Transparent Bridges in order to optimize data transfer paths and memory accessing by using their unique potential for mapping memory. Since 2018, Jonas has been working as a software architect for Dolphin Interconnect Solutions, continuing his work from his PhD. His research interests are distributed shared-memory applications, computer networks and cluster interconnects.

Lars Bjørlykke Kristiansen Lars Bjørlykke Kristiansen is a software architect at Dolphin Interconnect Solutions. He got his master's degree in Informatics at the University of Oslo, Norway in 2015 where his thesis laid the foundation for Device Lending. At Dolphin he continues his work on Device Lending, as well as exploring innovative new ways to exploit the unique shared memory capabilites of PCIe clusters and Non-Transparent Briding.

Rune Johan Borgli Rune Johan Borgli is a Ph.D. student at Simula Research Laboratory. He received his master's degree from the University of Oslo in 2018, where his master thesis topic was on hyperparameter optimization using Bayesian optimization on transfer learning for medical image classification. His research interests are machine learning workflows and pipelines, image processing, machine learning infrastructure optimization, and secure and privacy-oriented data handling. He is currently working on his Ph.D. thesis which will explore secure machine learning processing of privacy-sensitive data.

Håkon Kvale Stensland Håkon Kvale Stensland is a senior researcher at Simula Research Laboratory. He finished his master degree (MSc) in 2006 and received his doctoral degree (Ph.D.) in 2015 from the Department of Informatics, University of Oslo. At Simula, he is the deputy head of the Department of Advanced Computing and System Performance. From Simula, he is also leading the collaboration with Dolphin Interconnect Solutions, where we research sharing of GPUs and other IO devices between multiple machines connected in a PCI Express network. Håkon is also an adjunct associate professor at the University of Oslo, Department of Informatics, where he is involved in teachings and supervising Ph.D. and Master students.

Friedrich Seifert Friedrich Seifert obtained his master's degree in Computer Science (Dipl.-Inf.) from Chemnitz University of Technology, Germany, in 1999. He is working as Senior System and Software Architect for Dolphin Interconnect Solutions, where he focuses on developing innovative concepts for building compute and I/O clusters using Non-Transparent Bridging functionality found in state-of-the-art PCIe chipsets.

Michael Riegler Michael Riegler is a senior researcher at SimulaMet. He received his master's degree from Klagenfurt University with distinction and finished his PhD at the University of Oslo in two and a half years. His research interests are medical multimedia data analysis and understanding, image processing, image retrieval, parallel processing, crowdsourcing, social computing and user intent. He is involved in several initiatives like the MediaEval Benchmarking initiative for Multimedia Evaluation, which runs this year the Medico task (automatic analysis of colonoscopy videos). Furthermore he is part of an expert group for the Norwegian Council of Technology on Machine Learning for Healthcare reporting directly to the Norwegian Government.

Carsten Griwodz Carsten Griwodz is professor at the University of Oslo, Norway, and co-founder of ForzaSys AS, a social media startup for sports. He received his doctoral degree from Darmstadt University of Technology, Germany in 2000. His research interest is the performance of multimedia systems and his goal to understand how users can become sufficiently immersed in an experience depending on their goals and context. He explores research advances in fields ranging from operating system and networks to computer vision to understand and reach the point of sufficient immersion.

Pål Halvorsen Pål Halvorsen is a chief research scientist at SimulaMet, a professor at OsloMet University, a professor II at University of Oslo, Norway, and the CEO of ForzaSys AS. He received his doctoral degree (Dr.Scient.) in 2001. His research focuses mainly at distributed multimedia systems including operating systems, processing, storage and retrieval, communication and distribution from a performance and efficiency point of view. He also is a member of the IEEE and ACM.