vPHI: Enabling Xeon Phi Capabilities in Virtual Machines

Stefanos Gerangelos
Computing Systems Laboratory
National Technical University of Athens
sgerag@cslab.ece.ntua.gr

Nectarios Koziris
Computing Systems Laboratory
National Technical University of Athens
nkoziris@cslab.ece.ntua.gr

***Abstract*—Heterogeneous processing has gained popularity in the high performance computing (HPC) area lately and it appears to have a great potential for future data centers. In this regard, accelerators, such as GPUs and Intel Xeon Phi, have already started to play a significant role in HPC systems offering a high degree of parallelism to application developers. Furthermore, hardware virtualization is gaining interest in these domains as well, due to the benefits it offers, such as server consolidation, costs reduction and resource utilization. In this context, there is a growing potential for integrating accelerators into the virtualization stacks of the next generation.**

In this paper we take a first step towards enabling Intel Xeon Phi capabilities in virtual machines. We present vPHI, a framework for efficient virtualization of Intel Xeon Phi resources. To our knowledge, vPHI is the first approach that enables Xeon Phi sharing between multiple VMs running on the same physical node. We present vPHI as a proof-of-concept using QEMU-KVM as the hypervisor. vPHI utilizes Intel's SCIF API as the transport layer for transferring data over the PCIe to the accelerator device. Preliminary evaluation has shown promising results with vPHI being able to attain 72% of native execution in terms of peak throughput.

I. Introduction

Heterogeneous processing has gained popularity in the HPC context lately and it appears that it has a great potential for future data centers. As data creation worldwide keeps growing with remarkable rates [1], the processing power needs to be increased proportionally, in order to keep up with data creation rate. However, the multicore scaling trend presents a limit in the foreseeable future, which is referred in the literature as the dark silicon era [2, 3]. In this context, computer scientists and professionals estimate that the way data are processed will adapt to the new conditions and future data centers will gradually move from the scale-up paradigm to more heterogeneous architectures [4, 5].

Additionally, numerous studies have shown that, despite the large amount of processing power available in modern data centers, only a small portion of it is being utilized by the providers [6, 7]. One of the main reasons is that the interference between different workloads running in a same machine can in many ways severely hurt performance, despite the fact that the available processing resources are more than sufficient for the workloads to co-exist [8, 9]. Hence, providers prefer to underutilize some of their resources in favor of a more stable behavior with minimum performance variations for their clients. Researchers have proposed that moving to heterogeneous platforms can form a path to mitigate the consequences of this problem [4, 10]. One of the building blocks of heterogeneous computing ecosystem is accelerators, such as GPUs, Intel Xeon Phi or FPGAs.

At the same time, cloud computing has been established in many data-center infrastructures offering benefits both for the end users as well as the service providers, such as flexibility, server consolidation, costs reduction and resource utilization among others. With the aforementioned potential that heterogeneous computing and accelerators appears to develop, there is a growing need for integration in the current cloud stacks. In essense, virtualization-aware systems need to embrace accelerators by adapting their components into the specialized nature of this kind of hardware.

In recent years, there are efforts to enlarge the application scope of accelerators inside Virtual Machines (VMs). Interesting research works have been published mostly targeting GPUs [11]–[19]. Additionally, in the enterprise section, companies have started to offer GPUs as a service in a cloud context [20, 21]. Based on our experience of accelerating applications in virtualized environments, we argue that system engineers have to design next generation accelerator technologies bearing in mind the need of device integration on virtualized environments both in the software as well as the hardware level.

Concerning Xeon Phi coprocessors family, Intel offers a couple of solutions with KVM [22] and Xen hypervisor [23] using the *PCIe passthrough* method. This virtualization approach offers near-native performance at the expense of inability to share a single accelerator device to many VMs, since the Xeon Phi card is being directly assigned exclusively to a single VM.

To our knowledge, there is currently no solution enabling Xeon Phi sharing between virtual machines in a same host. In this paper, we make a first step towards Xeon Phi virtualization and sharing between VMs. We propose vPHI, a low overhead Xeon Phi virtualization framework, that accelerates virtual machines by enabling them to execute code on a Xeon Phi card. We implement vPHI using QEMU-KVM [24, 25] as a proof-of-concept, but the concept can be applied to other virtualization environments as well. vPHI provides virtualization of Intel's SCIF (Symmetric Communication Interface),

which is the transport layer that is used between the host and the accelerator devices over the PCIe bus. vPHI is binary-compatible with precompiled applications, alleviating the need for porting or even recompiling existing source code. vPHI supports all three modes that are defined in the Xeon Phi model of execution, i.e. *native*, *offload* and *symmetric*. Preliminary results in native execution mode show promising perspectives for Xeon Phi sharing and integration in current cloud stacks.

The rest of the paper is organized as follows: we provide the necessary background in Section II. Section III describes the design and implementation of vPHI, while section IV presents performance evaluation results. In Section V we discuss related work and finally in Section VI we conclude and present directions of future work.

II. Background

Intel Xeon Phi is a product family of processors that employ Intel's MIC (Many Integrated Core) architecture. It consists of a series of massively-parallel manycore processors that provides x86 compatibility and can be used to accelerate a system as a coprocessor, or even as a host processor (e.g. Knights Landing). In this paper, we target Xeon Phi coprocessor case and provide a solution that can be used in virtualized environments. Next, we provide some background details about the technologies and the software stacks that we use.

A. Xeon Phi model of execution

Intel defines a model of execution, including three modes to meet different use-case needs: *native*, *offload* and *symmetric* modes of execution. In *native* mode the user supply the executable directly on the Xeon Phi card. *Offloading* mode permits the user to execute the application on the host CPU and offload some compute-intensive workloads to the coprocessor using the corresponding directives of a framework, e,g, *OpenMP*. Finally, in *symmetric* mode Xeon Phi can be viewed as an independent node and in that way a user can launch some processes of the same parallel application on the host side and some other processes on the accelerator, using for example *MPI*. In this work, we evaluate vPHI using the native mode of execution. However, vPHI supports all three modes, since all of them utilize SCIF as the transport layer to communicate with the device.

B. Xeon Phi system software stack

Current Xeon Phi devices are connected to the system through the PCIe bus. Intel provides SCIF (Symmetric Communication Interface), a low-level abstraction layer over PCIe, in order to enable higher-level components to exploit DMA capabilities of Xeon Phi without messing directly with PCIe transactions. With vPHI, we essentially provide a virtualization scheme of SCIF to enable the same functionality for VMs. Figure 1 depicts the general system architecture of the software stack that is used in a typical non-virtualized Xeon Phi environment. Using SCIF, applications running in the host as well as the device can communicate with each other using the same API. In order to achieve that, Xeon Phi itself boots a micro operating system (*uOS*), which consists of a modified Linux kernel, that includes a SCIF driver. Also, to expose the SCIF API, a SCIF library (libscif) is used. Furthermore, Xeon Phi software stack includes an emulated network driver as part of the uOS, that uses SCIF, and enables users to utilize network tools (e.g. `ssh`) and remotely connect to the Xeon Phi device. In this way, they can execute applications on the coprocessor using a shell. Similar to the Xeon Phi card, the respective components, libscif and SCIF driver, have been implemented for the host side.

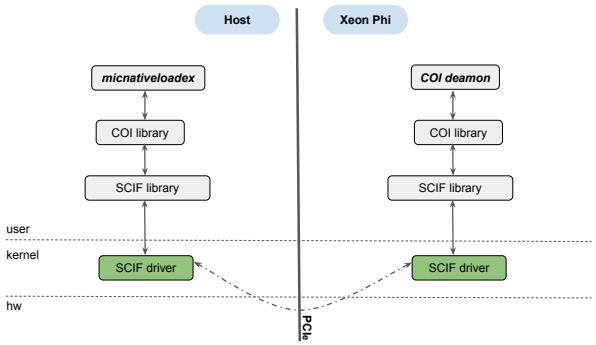

Fig. 1: Communication between host and Xeon Phi

The library exposes the SCIF API to the application-side and communicates with the SCIF driver by performing system calls to a character device, namely `/dev/mic/scif`. SCIF API provides both one-way and two-way communication semantics. There is a family of socket-like SCIF calls (`scif_bind()`, `scif_listen()`, `scif_accept()`, `scif_connect()`, `scif_send()`, `scif_recv()`) that supports traditional send-receive communication and another set of calls (`scif_register()`, `scif_unregister()`, `scif_(v)readfrom()`, `scif_(v)writeto()`, `scif_mmap()`, `scif_munmap()`) that exposes read/write (RDMA) semantics. RDMA is a common communication pattern used in high performance interconnects domain. In such contexts, developers frequently use a combination of RDMA and polling as an alternative to blocking methods, in order to notify the client of an I/O completion event. Similarly, SCIF provides `scif_poll()` to inform the caller that a subsequent operation to a specific endpoint can be performed without blocking, which for example could mean that some data have been received. Finally, there is another set of SCIF calls (`scif_fence_*()`) that act as synchronization barriers. vPHI provides the same interface to the applications running in VMs by redirecting the traffic through the host.

However, for certain use cases even SCIF exposes unnecessary details that many runtime systems or libraries do not need to be aware of. Hence, Intel provides a higher-level library which uses SCIF as the transport layer and abstracts the low-level details (Figure 1). This library is called COI (Coprocessor Offload Infrastructure) and can be used to build runtime frameworks in order to query and control the state

of Xeon Phi devices available in the system or to offload computational workloads to the coprocessor, by loading the appropriate libraries and executables, transferring the data over PCIe. Intel has implemented a set of tools for this purpose, which are included in their platform software stack (Intel MPSS [26]). We use on of these tools (`micnativeloadex`) to evaluate our framework in native mode of execution. Xeon Phi device receives the respective requests from the host through a COI daemon that is executed after uOS has booted. By virtualizing SCIF transport layer, vPHI remains compatible with higher-level frameworks, such as COI.

C. I/O Virtualization software stack

We implement vPHI in QEMU-KVM environment using the paravirtualization approach. Paravirtualization enables low overhead I/O virtualization by establishing an efficient communication channel between the host and the VM (guest). Using this method the virtual hardware exposes a software interface to the respective driver in the guest, which is aware that is being virtualized and thus reduces any unnecessary I/O traffic that a virtualization unaware driver would produce. The mechanism we use for this purpose is *virtio* [27], which is a standardized interface used for development of virtualized devices, as well as a mechanism to support communication between the guest and the hypervisor.

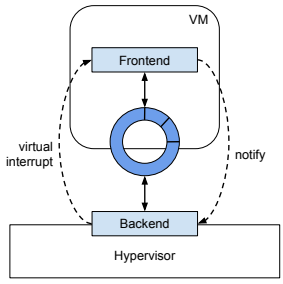

Fig. 2: Virtio transport mechanism

Virtio follows the split-driver model approach, according to which a paravirtualized frontend driver is inserted into the guest, communicating with the respective backend on the host side. For the communication to be realized a shared ring structure is registered between the guest and the host (Figure 2). The frontend driver submits I/O requests by posting the respective buffers in the shared ring and notifying the backend. Afterwards, the backend processes the event, emulates the I/O that is requested and produces a corresponding response. It posts the response in the ring and notifies the guest side via a virtual interrupt. Guest can either busy-wait on the shared ring, consuming CPU cycles, or block until the request is accomplished. In the latter case, the interrupt produced by the host, wakes-up the guest, which pushes up the response to the upper layers. We point out here that no copies are involved during the communication between the guest and the host, since a shared memory area (ring) is used and also the host can access guest's physical address space and map the corresponding buffers to its own address space.

III. Design and Implementation

We implement vPHI as a proof-of-concept using QEMU-KVM as the hypervisor and virtio as the paravirtualization interface. As shown in Figure 3, vPHI consists of a guest kernel driver and a QEMU backend device implemented in host user space. We also need to make a slight modification to the kvm host kernel module, in order to properly redirect page faults to the guest. In general, we try to be as less intrusive as possible. In this context, the QEMU backend device and the guest kernel driver can be applied without host kernel disruption. Nevertheless, the previously mentioned modification to the host kernel kvm driver is necessary in order to support mapping of guest user memory areas to Xeon Phi device memory through `scif_mmap()`. We further analyze this on the following paragraphs.

Essentially, the main role of vPHI is to intercept original SCIF transport requests and to redirect them through the backend QEMU device. Upon receiving these requests, the backend driver forwards them to the host SCIF driver, which controls the physical device. After the completion of an I/O request, the results are pushed back to the stack following the opposite direction eventually reaching the original requester, which usually is a runtime's transport layer. Simultaneous multi-threaded execution requests from different VMs can end up running in parallel on the Xeon Phi device spreaded across the available cores of the card. If there is an oversubscription considering requested threads to physical cores ratio, then the resource multiplexing is accomplished by the scheduler of the uOS which runs on a dedicated Xeon Phi core. In Figure 3 we consider a SCIF request triggered by an application inside a VM and show the corresponding I/O path. This is a representative scenario that occurs in any of the three aforementioned Xeon Phi modes of execution. Solid lines represent control path, while dashed lines represent data path. We refer to each phase of Figure 3 in the subsequent paragraphs, as we describe each component and how it operates.

Based on Figure 3 we describe the example of an application launched on the Xeon Phi accelerator from a VM using `micnativeloadex` from Intel MPSS [26]. This scenario can occur in parallel with different applications inside a VM or even by multiple VMs using vPHI, which enables sharing at both levels. The tool has to request DMA transactions with Xeon Phi as the destination for the binary, libraries etc., through (`libscif`) library (3a). Afterwards, `libscif` issues (3b) the corresponding system call (`open()`, `close()`, `ioctl()`, `poll()`, `mmap()`) depending on the operation requested. Most of the SCIF functionality is exposed to user space through different `ioctl()` commands. Since vPHI implements SCIF operations, both the application-tool as well as `libscif` remain intact and no recompilation is even needed. Hence, the issued system call is intercepted by the vPHI frontend driver.

vPHI frontend driver: We implement vPHI frontend driver as a Linux kernel module which is inserted dynamically at guest kernel space. The driver acts as a "glue" between

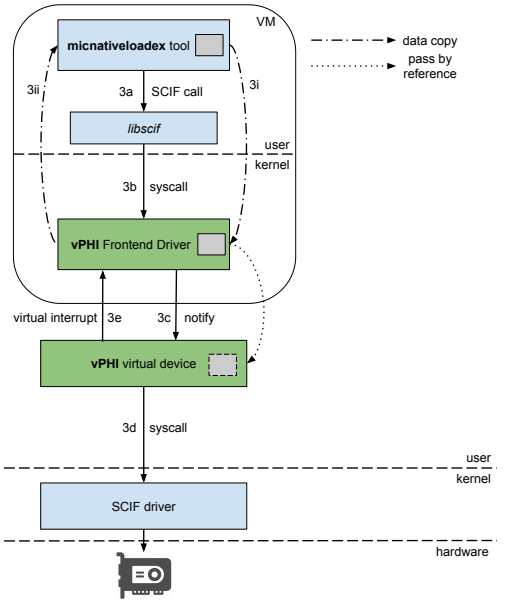

Fig. 3: vPHI Architecture (Data and Control Path)

virtualization-unaware `libscif` and the rest of the stack by forwarding the operations requested to vPHI backend device through virtio communication channels. Among its duties, the frontend driver multiplexes requests and orchestrates the user space threads or processes that are waiting for a response from the coprocessor. We had two design choices in this step: we can either implement a polling-based method or an interrupt-based one. Since busy-waiting on a shared resource consumes CPU cycles, we choose the interrupt-based approach, adding up some extra overhead when the driver sets up the sleeping mechanism, in favor of better performance when the number of parallel requests increases. Thus, the driver places a reference to a buffer in the shared ring structure, then notifies the back-end device (3c) that there is a pending request and registers the waiting mechanism until a wakeup event arrives. When this happens, the interrupt handler checks the last response in the shared ring and wakes up the respective entity to continue and push the data up to the stack. Throughout this procedure the only copies that occur are the ones between user space and kernel space at each direction of the path (3i, 3ii). Every other data exchange is realized through references reducing the virtualization overhead especially for large data transfers.

vPHI backend device: We design vPHI backend device as a virtual PCI device and implement it as a QEMU extension. As we mentioned previously, the backend is notified by the frontend (3c) when a new request has been pushed to the virtio ring. Then, the backend checks the shared ring and maps the buffer to its address space avoiding again any copies. The backend has access to the memory mappings of guest hardware address space to host user space since it registers guest memory when the VM boots. Afterwards, the backend performs the relevant system call (3d) to the host SCIF driver and waits for the result. When the system call returns, it pushes the result in the shared ring and notifies the guest via a virtual interrupt (3e). Following this approach, every VM on the same physical machine, is represented by a different QEMU host process. Thus, Xeon Phi sharing is enabled, as from the host driver's perspective, multiple VMs issuing SCIF request are essentially multiple host processes that execute system calls to SCIF driver in parallel.

Blocking vs non-blocking mode: QEMU has been implemented based mainly on an event-driven approach. As such, QEMU handles events as they are produced and during that time the whole VM is in blocking mode. Any previously running entity inside the guest pauses. This model prevents race conditions and avoids many synchronization points at the cost of suspending the execution of the virtual machine. That way, event handling should complete as early as possible to prevent noticeable pauses of the guest. In a few cases, when this is not possible, QEMU follows a threading model, according to which it spawns a worker thread that executes the long-running handling of the event, and falls back to the event-driven mode unfreezing the VM.

In designing vPHI functionality, we had to decide between these two modes for SCIF operations. Following QEMU's approach, we choose the blocking mode for most SCIF operations and a non-blocking mode for operations that otherwise would potentially block the virtual machine for an unacceptable period of time. For example, SCIF defines a connection-oriented model between two endpoints before data transfers begin and in this context it implements `scif_listen()` and `scif_accept()` with similar logic to the POSIX sockets API. Hence, we implement `scif_accept()` in a non-blocking way, since we do not know beforehand when a corresponding `scif_connect()` request will arrive. For data transfer requests, we follow the blocking model, although one can argue that the blocking cost increases proportionally with the data size. The performance tradeoff to consider here arises on the one hand from the blocking cost that prevents any other threads inside the guest to make progress, while on the other hand from the overhead of creating and eventually destroying the worker thread. As the data size increases, the non-blocking method appears more appealing. Thus, as a future direction, we plan to implement a hybrid model, according to which vPHI will use the blocking method for smaller data transfers, while for larger ones, it will switch to the non-blocking approach.

Guest memory registration and MMIO: Apart from two-way send-receive communication semantics, SCIF also supports remote memory access for exchanging data between host and device memory, exposing the relevant API (`scif_(v)readfrom()`, `scif_(v)writeto()`). For a buffer to be involved in a set of remote memory operations, the relevant memory pages have to *pinned*. Memory pinning refers to a procedure according to which a page or a set of pages are marked in a way that prevents the operating system from swapping them out. In this way, a subsequent remote memory **read** from these pages would load valid data to the remote node. If the respective pages are not pinned, and happens to have been swapped out, the read operation

will acquire invalid data, without any chance to produce a page fault and bring back the original data from the disk. Likewise, a remote memory **write** operation could overwrite data of some other process in case of a previous swap out. SCIF exposes the memory pinning functionality through `scif_register()`/`scif_unregister()` calls. In vPHI implementation of memory pinning we first pin the pages that the user requested in the guest operating system. These pages correspond to the user-supplied buffer which is referred to the guest user space. However, the buffers which are pushed to the shared ring are always referred to the guest kernel space and subsequently, virtio translates them to guest physical space in order to be later accessed by QEMU backend through the mapping to its own host user space. Thus, before it uses the shared ring, vPHI first maps the respective pinned page to a kernel address, and then it pushes this buffer to the virtio ring for further processing.

Apart from remote read/write operations with the pinned buffers, SCIF supports memory mapping of remote buffers to the local address space through `scif_mmap()`. After a successful call to `scif_mmap()`, in order to access Xeon Phi memory, the user can simply dereference (load/store) the relevant `mmap`'ed memory address without any intermediate library or system call. This memory access would either page fault to the host operating system, which will confirm the validity of the mapping and fetch the corresponding frame to main memory or retrieve the data from the main memory that a previous memory access has forced to fetch. Inside `scif_mmap()` host properly setups the corresponding memory management structures pointing to device memory. In vPHI, we perform a two-level mapping, one from the user-supplied address to a guest physical frame and a second from the guest physical frame to the host physical frame, which corresponds to Xeon Phi memory. The problem with this approach lies in the fact that if an application running inside a virtual machine performs e.g. a pointer dereference of a previously successfully mapped buffer, then it will fault into kvm host kernel module, which will try to examine the situation based on the address that faulted. However, this address will be interpreted by the host driver as a reference to its own address space leading to an invalid memory area. In order to overcome this problem, we have to make a slight modification to the host driver as well as the kvm. Linux kernel separates different mappings by defining different *vmas* (virtual memory areas). We therefore tag every vma that has been created by vPHI during `scif_mmap()` using a new label (`VM_PFNPHI`) and store the relevant physical frame number. Then, in every fault that is triggered by a vPHI `mmap`'ed area, kvm spots the frame number that corresponds to the respective Xeon Phi memory region. The modifications in terms of lines of code are slight (less than 10 LOC in `kvm` and less than 15 LOC in host SCIF driver).

Implementation details: During a SCIF data transfer using vPHI, the guest frontend driver first allocates a buffer and copies the user-supplied data (for the send/write case) or the received data (for the receive/read case). In this step we want to allocate guest physically contiguous pages, so we use kernel's `kmalloc()` API, since this set of pages will be later used for I/O between guest and host through virtio ring. However, Linux memory subsystem imposes a limitation on the maximum set of physically contiguous pages that can be allocated. This upper limit is defined in the kernel (`KMALLOC_MAX_SIZE`) and depends on the architecture. Specifically, for `x86_64` architecture the limit is 4MB. Hence, if the requested data size is greater than this value, we implement the data transfer breaking up the allocation to `KMALLOC_MAX_SIZE` elements and proceed with each one of them.

Host Xeon Phi driver exposes a set of information related to the Xeon Phi, such as the family codename of the accelerator, through the `sysfs` filesystem. Some of Intel's MPSS software runtimes and tools, including `micnativeloadex`, rely on this information to operate as intended. Thus, we implement the necessary functionality that is required, in order to be able to successfully launch a MIC executable on the Xeon Phi, and we expose the same information that is provided in the host.

IV. Performance Evaluation

A. Experimental Setup

In this section we describe the experiments we performed to analyze the behavior of vPHI and we present the corresponding results. We setup a host machine with 1x Intel Xeon E5-2695 v2, 64GB RAM (DDR3-1600Mhz), also equipped with one Intel Xeon Phi 3120P coprocessor. We configure host as a VM container using QEMU-KVM (version 2.2.50) as the hypervisor.

At first, we implement a set of microbenchmarks to evaluate SCIF performance and we show the results in the subsection IV-B. Next, we conduct a higher-level experiment using `dgemm` from Intel samples [28] to multiply matrices. For the `dgemm` experiment we follow the *native* mode of execution according to Intel's execution model.

In native mode of execution there are two choices. The user can either `ssh` to the accelerator and execute the application locally, or launch the MIC executable directly from the host. In the first case the user should explicitly copy the executables, libraries and other dependencies on the coprocessor and then execute the application. In a virtualized environment, this can become possible by configuring a network bridge on the host between the emulated `mic0` network interface and the interface that is attached to the VM. However, this configuration is not well-suited for cloud environments. Such setups can end up with many users logged in a shared accelerator environment ruining the isolation characteristics of cloud computing. Hence, we test native mode using the latter case described, which is enabled by vPHI.

B. Microbenchmark Performance

We implement a set of microbenchmarks that exchange data over the PCIe between the host and Xeon Phi using SCIF. We execute these benchmarks in order to estimate the virtualization overhead of vPHI. We analyze vPHI performance

using send-receive two-way communication as well as remote memory operations. First, we execute the benchmark on the host, in order to obtain the baseline performance. Then, we spawn a single-core VM with vPHI and execute the benchmark in the virtualized environment. In both cases, a corresponding server is executed on the coprocessor, in order to serve SCIF send request (in the send-receive case) or properly register device memory (in the remote memory case).

In order to measure latency, we use the send-receive benchmark, according to which a SCIF server is launched on the accelerator, listens for connection requests and when a connection is established, it blocks on `scif_recv()`, waiting to serve data to the respective client. In this context, a SCIF client is executed on the host (or on the VM), which connects to the server and sends a number of data. We show the corresponding latency measured for the host as well as for the vPHI in Figure 4.

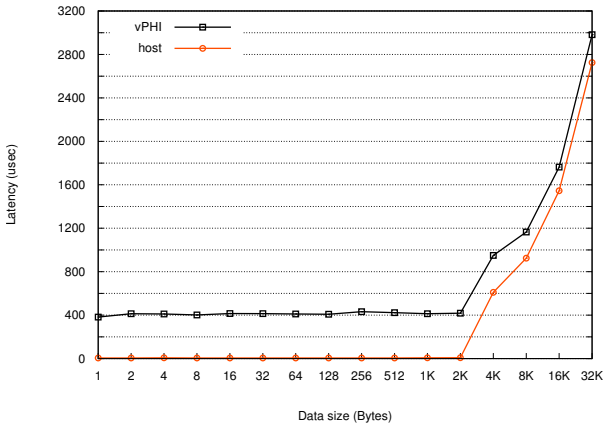

Fig. 4: Send-receive communication latency

For the native (`host`) execution the latency for sending 1 Byte is 7 us, while for the virtualized one, the respective latency climbs up to 382 us. Hence, the virtualization overhead of vPHI is 375 us (=382−7). Since this is a notable increase, we performed deeper breakdown measurements to further investigate the cause of this overhead. Based on the breakdown analysis, we conclude that 93% of this overhead attributes to the waiting scheme of vPHI inside the frontend driver. More specifically, when the frontend driver issues a SCIF request to the shared ring, the relevant process is placed on a waiting queue, until the request is fulfilled. When the backend has finished the execution of the request, it triggers a virtual interrupt and the interrupt handler in the guest wakes up all sleeping processes, which check the shared ring to determine if the reply is for them. The mechanism of sleeping and waking up is the main source of performance degradation for latency-sensitive workloads. As we mentioned in the previous sections, this scheme is necessary for larger data transfers in order to reduce the CPU utilization of an alternative busy-wait method. However, we plan to implement a hybrid approach that uses each time the best of the two available schemes depending on the requested data size, so we can enable near-native latency for small data sizes, while retaining acceptable transfer rate for larger ones. Finally, in Figure 4 we can observe that the previously mentioned overhead remains constant as data size increases, so there is a constant offset in terms of latency compared to the baseline measurement.

Next, we execute another benchmark using the SCIF remote memory access model, which is more suitable for larger data transfers, in order to estimate the maximum throughput that vPHI can attain. In this experiment, we launch an executable on Xeon Phi, that again listens for incoming connections and then pins a device memory area based on the requested size using `scif_register()`. In the host (or VM) side the benchmark requests a connection and afterwards it performs a remote read from the accelerator device. The results are depicted in Figure 5. We can observe that host remote read can reach 6.4GB/s, while vPHI's respective throughput is 4.6GB/s, which equals to 72% of the host case.

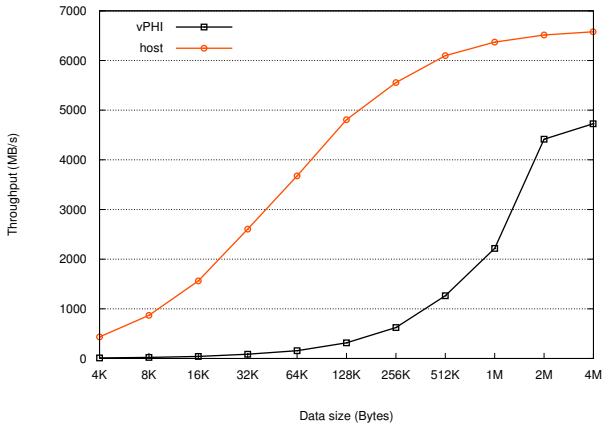

Fig. 5: Remote memory access throughput

C. Application Performance

In this subsection we show the results of a higher-level application that we used with vPHI. We measured the execution of `cblas_dgemm` matrix multiplication from Intel samples [28], which uses the MKL [29] library. We use `micnativeloadex`, a tool that Intel provides to enable launching of MIC executables to the coprocessor directly from the host, following the native mode approach. As we previously described, `micnativeloadex` uses COI library and communicates using the SCIF protocol with `coi_daemon` executed on the coprocessor. `micnativeloadex`'s role is to properly setup the environment, launch the necessary libraries and executables and spawn the requested number of threads.

In this experiment we execute `micnativeloadex` with `dgemm` as the supplied binary on the host and on the VM. After the moment that `dgemm` executable has been launched on Xeon Phi and since it is executed as a whole without vPHI intervention, we observed no performance degradation for the vPHI compared to the host concerning actual execution time on the device. In order to estimate the overhead of vPHI in the

entire offloading procedure, however, we also measure the total time of execution from the moment that `micnativeloadex` is launched on the host (or the VM) until the final results are produced and the tool finishes execution. We vary the number of threads as well as the size of the matrices and plot the results in Figure 6, Figure 7 and Figure 8 for 56, 112 and 224 number of threads respectively.

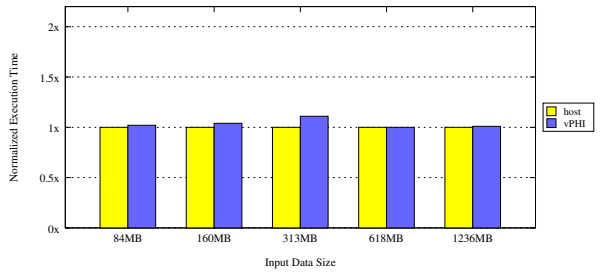

Fig. 6: Launch and execution of `dgemm` using 56 threads

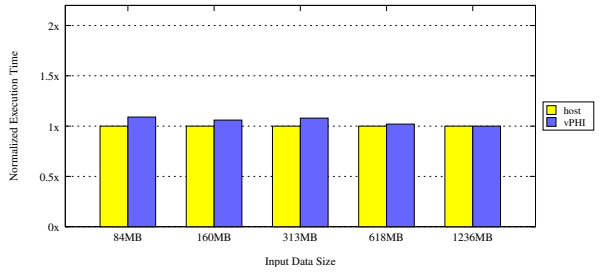

Fig. 7: Launch and execution of `dgemm` using 112 threads

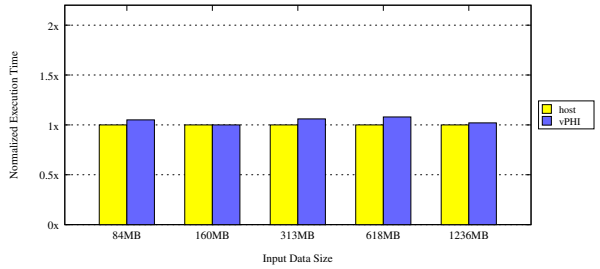

Fig. 8: Launch and execution of `dgemm` using 224 threads

The Y axis represents the normalized total time of execution that includes the launching of the necessary binaries using `micnativeloadex` from the host (or the VM) and the actual execution on the accelerator. The X axis represent the total size of the two input arrays. We try to setup and present a meaningful experiment in terms of input data size. From the above figures we can draw the conclusion that for larger experiments (order of seconds), which include longer-running loops as well as transferring sizable binaries (libraries/executables) over the PCIe, the virtualization cost of vPHI is amortized and the relative overhead compared to the total execution time is negligible. In contrast, as the size of transfered data decreases, vPHI's virtualization overhead has a greater impact, as the previous latency experiments prove. However, given the cost of a PCIe transaction, a typical scenario involving a coprocessor usually consists of loading a substantial amount of data following by a heavy computational phase, because otherwise it may not worth the effort of offloading a small amount of data and perform a light computation that could be carried out on the local CPU with less overhead.

V. Related Work

Xeon Phi consists of a relatively new and evolving architecture family. In order to enable access from a VM, Intel [22, 23] and VMware [30] use the *PCIe passthrough* technique to directly assign the coprocessor to a single VM. Although, using this method an application running in a virtual machine can perform at a near-native rate, it does not support sharing of one physical device to multiple VMs. To our knowledge, there is currently no solution that enables sharing of a Xeon Phi coprocessor to multiple virtual machines. Furthermore, ScaleMP provides to the application a virtual pool of cores and memory by aggregating resources from the host and the accelerator in a unified manner [31]. This approach eventually provides a large SMP configuration to the user rather than a setup that uses the offload model of execution.

Considering the problem of accelerators virtualization, many approaches have been proposed targeting mostly GPUs. In the meantime, many cloud providers, such as Amazon [20], Microsoft Azure [21] etc. have started to offer GPU computing resources as a service. We briefly mention GPU virtualization approaches here, as we propose a virtualization solution for a different accelerator with different characteristics. However, approaches from the GPU virtualization domain are included in the set that motivates us for this work. Similar to the Xeon Phi case, a class of proposed solutions includes the use of passthrough technology to provide to a VM direct access to a host device. Additionally, there are solutions [16, 32] that expose a static number of virtualized GPUs, each one of them is directly assigned to a virtual machine. Other works fall into the full virtualization category [15], according to which no modification is performed to the guest operating system. Some approaches have been proposed using the paravirtualization technique [11]–[13], while others employ both full virtualization and paravirtualization methods [14]. API redirection technique is used by some solutions in order to provide GPU virtualization features [17] or to even remotely execute GPU jobs [18]. In the context of remote execution, there are efforts [19] to optimize the intra-node communication between VMs located on the same host and thus to execute remote GPU frameworks with lower overhead. Finally, there are some approaches towards providing virtualization solutions for FPGAs, targeting either a better exploitation of the underlying hardware by different sections of an application [33] or the integration to cloud stacks [34].

VI. Conclusion

In this work we present vPHI, a framework for efficient virtualization of Intel Xeon Phi resources and sharing between VMs running on the same host. We design vPHI following

the split-driver model, with a frontend driver running in the guest operating system and the respective backend as a QEMU process in the host userspace. vPHI provides transparency and binary compatibility with existing precompiled MIC executables, since it operates at the transport layer and complies with Intel's SCIF low-level API. Experimental evaluation of our prototype reveals that vPHI has negligible overhead when combined with large data transfers over the PCIe, which are present in many typical coprocessor use cases.

In this work we evaluate our framework using the *native* mode of Xeon Phi execution model. However, since vPHI supports all three execution modes, we plan to thoroughly evaluate vPHI in *offload* and *symmetric* mode of execution as a future work, using popular frameworks, such as OpenMP and MPI respectively. Finally, future endeavors also include optimizing further our framework by implementing hybrid models that will combine blocking and non-blocking mode depending on the data size and deeply investigating the guest driver's waiting scheme, in order to also enable potential latency-sensitive applications to benefit from vPHI.

REFERENCES

[1] In Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 20142019 White Paper.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Power challenges may end the multicore era," *Commun. ACM*, vol. 56, no. 2, pp. 93–102, Feb. 2013.

[4] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, July 2011.

[5] M. Shafique, S. Garg, T. Mitra, S. Parameswaran, and J. Henkel, "Dark silicon as a challenge for hardware/software co-design: Invited special session paper," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES '14. New York, NY, USA: ACM, 2014, pp. 13:1–13:10.

[6] L. A. Barroso, "Warehouse-scale computing: Entering the teenage decade," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, Jun. 2011.

[7] L. A. Barroso and U. Hoelzle, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[8] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 248–259.

[9] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 77–88.

[10] C. Kachris, G. Gaydadjiev, H. N. Nguyen, D. S. Nikolopoulos, A. Bilas, N. Morgan, C. Strydis, V. Spatadakis, D. Gardelis, R. Jimenez-Peris, and A. Almeida, "The vineyard project: Versatile integrated accelerator-based heterogeneous data centres," in *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, May 2016, pp. 1–4.

[11] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A gpgpu transparent virtualization component for high performance computing clouds," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, ser. EuroPar'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 379–391.

[12] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa, "Logv: Low-overhead gpgpu virtualization," in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov 2013, pp. 1721–1726.

[13] D. Vasilas, S. Gerangelos, and N. Koziris, "Vgvm: Efficient gpu capabilities in virtual machines," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 637–644.

[14] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Why not virtualizing gpus at the hypervisor?" in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 109–120.

[15] K. Tian, Y. Dong, and D. Cowperthwaite, "A full gpu virtualization solution with mediated pass-through," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 121–132.

[16] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class gpu resource management in the operating system," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 37–37.

[17] L. Shi, H. Chen, J. Sun, and K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, June 2012.

[18] A. J. Pea, C. Reao, F. Silla, R. Mayo, E. S. Quintana-Ort, and J. Duato, "A complete and efficient cuda-sharing solution for {HPC} clusters," *Parallel Computing*, vol. 40, no. 10, pp. 574 – 588, 2014.

[19] A. Nanos, S. Gerangelos, I. Alifieraki, and N. Koziris, "V4vsockets: Low-overhead intra-node communication in xen," in *Proceedings of the 5th International Workshop on Cloud Data and Platforms*, ser. CloudDP '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:6.

[20] Amazon GPU Instances, http://aws.amazon.com/ec2/instance-types/.

[21] "Microsoft Azure," https://azure.microsoft.com/en-us/.

[22] "Getting Kernel-Based Virtual Machine (KVM) to Work with Intel Xeon Phi Coprocessors," https://software.intel.com/en-us/articles/getting-kernel-based-virtual-machine-kvm-to-work-with-intel-xeon-phi-coprocessors.

[23] "Getting Xen working for Intel Xeon Phi Coprocessor," https://software.intel.com/en-us/articles/getting-xen-working-for-intelr-xeon-phitm-coprocessor.

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liquori, "kvm: the linux virtual machine monitor," in *Linux Symposium*, Ottawa, Ontario, Canada, 2007, pp. 225–230.

[25] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[26] "Intel Manycore Platform Software Stack (Intel MPSS)," https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[27] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008.

[28] "Intel Software Product Samples and Tutorials," https://software.intel.com/en-us/product-code-samples?value=20825&value=20802.

[29] "Intel Math Kernel Library (Intel MKL)," https://software.intel.com/en-us/intel-mkl.

[30] "Using the Intel Xeon Phi Compute Accelerator with ESX 6.0," https://cto.vmware.com/using-intel-xeon-phi-esx-6-0.

[31] "Accelerated-Computing - ScaleMP," http://www.scalemp.com/solutions/accelerated-computing.

[32] Nvidia, "NVIDIA GRID Virtual GPU Technology," http://www.nvidia.com/object/grid-technology.html.

[33] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker, "A heterogeneous multicore system on chip with run-time reconfigurable virtual fpga architecture," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 143–149.

[34] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 109–116.