2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era

Knapp: A Packet Processing Framework for Manycore Accelerators

Junhyun Shim
SAP Labs Korea
Email: junhyun.shim@sap.com

Joongi Kim
Lablup Inc.
Email: joongi@lablup.com

Keunhong Lee
School of Computing
KAIST
Email: khlee@an.kaist.ac.kr

Sue Moon
School of Computing
KAIST
Email: sbmoon@kaist.edu

***Abstract*—High-performance network packet processing benefits greatly from parallel-programming accelerators such as Graphics Processing Units (GPUs). Intel Xeon Phi, a relative newcomer in this market, is a distinguishing platform because its x86-compatible vectorized architecture offers additional optimization opportunities. Its software stack exposes low-level communication primitives, enabling fine-grained control and optimization of offloading processes. Nonetheless, our microbenchmarks show that offloading APIs for Xeon Phi comes in short for combining low latency and high throughput for both I/O and computation. In this work, we exploit Xeon Phi's low-level threading mechanisms to design a new offloading framework, *Knapp*, and evaluate it using simplified IP routing applications. Knapp lays the ground for full exploitation of Xeon Phi as a packet processing framework.**

I. Introduction

Recent advances in software routers built on high-end commodity hardware have extracted maximum hardware performance in IPv4/v6 routing [1]. Such systems reach the hardware upper bound through diverse software-based techniques, such as batching and kernel-bypassing [2], [3], [4], [5], [6], [1], [7]. On the hardware end, commodity NICs have advanced in capacity to transmit and receive 100 Gbps on a single port. On top of these advancements, parallel-programming accelerators such as GPUs have proven to match or significantly outperform CPU-only implementations of software routers [1], [5], [4].

While GPU is widely used as a co-processor in supercomputing, Intel Xeon Phi is a relatively recent addition to the pool of general-purpose co-processors but already is in the first place on the Top 500 supercomputers list as of November 2016 [8]. Out of the Top 500 supercomputers, 68 of them use NVIDIA GPUs and 34 use Intel Xeon Phi. The most commonly used architecture code-named *Knights Corner* has up to 61 Intel Atom cores with the core frequency around 1 GHz and on-board GDDR5 memory, and runs a custom branch of Linux OS itself [9]. With reported sequential memory bandwidth of 320 GB/s and 240 hardware threads, each capable of executing up to 16 operations in a single instruction, Xeon Phi is a strong contender for GPGPUs (general-purpose GPUs). Aside from the architectural differences from GPUs, Xeon Phi offers transparent code control in stark contrast to GPUs. Host-device communication and code execution on NVIDIA GPUs is only possible via NVIDIA's proprietary CUDA (Compute Unified Device Architecture). Xeon Phi has made public all the source code of its custom Linux and PCIe communication drivers, opening up a wide range of choices in application design.

The goal of this work is to explore the opportunity of Xeon Phi as a packet processing accelerator.

As a first step, we formulate core performance metrics for packet processing: integer operations throughput, random memory access rate, and thread synchronization overheads. We perform a set of microbenchmarks measuring those metrics of Xeon Phi against CPUs and GPUs. To ensure that computing power is not the bottleneck, we draw up an optimistically estimated throughput assuming ideal conditions (e.g., free of stalls from data dependency and full vectorization) for both Xeon Phi and GPU. This preliminary evaluation shows that Xeon Phi delivers comparable performance to CPUs and GPUs.

As the next step, we search for an efficient computation offloading mechanism for Xeon Phi. Our analysis and experiments show that offloading primitives in existing runtimes such as OpenCL and OpenMP offer sub-optimal performances for packet processing. Both runtimes show asymmetric data transfer latency between CPUs and Xeon Phi as well as increased execution time for offloaded code segment. Based on this observation, we design a custom offloading scheme mechanism the low-level threading and data transfer primitives: running our own multi-threaded packet processing daemon on Xeon Phi built on top of SCIF (symmetric communication interface) and POSIX threads (pthread).

Finally, we build and evaluate an offloading framework named *Knapp* that runs simplified IPv4 and IPv6 routers. It receives packets from networks, processes them in the Xeon Phi processor, and forwards them. To balance parallelism and synchronization overheads, we devise a novel scheme called `vDevice` that serves as a virtual split view of the Xeon Phi processor, comprised of consecutive logical cores with its own thread groups. It reaches up to 22 Gbps with minimum-sized packets and 40 Gbps with larger packets on a single socket configuration with four 10 GbE ports. To better exploit Phi's wide SIMD units, we also write hand-vectorized variants of IPv4 and IPv6 routing applications. We find that fully vectorizing our applications improves the per-batch processing latency by 2×, achieving higher forwarding throughput with less Xeon Phi cores. When utilizing sufficient number of Xeon

978-1-5090-6354-3/17 $31.00 © 2017 IEEE
DOI 10.1109/HiPINEB.2017.8

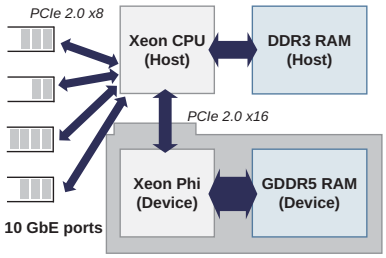


Fig. 1. Block diagram of a single-socket Xeon Phi configuration with a Xeon CPU and four 10 GbE cards.

Type	Model	# of cores	Memory B/W	Random Access Rate
CPU	Intel Xeon E5-2670	8	51.2 GB/s	563 M/s
GPU	GTX 680	1,536	192 GB/s	946 M/s
GPU	GTX 980	2,048	224 GB/s	1,263 M/s
MIC	Intel Xeon Phi 5110P	60	320 GB/s	500 M/s / 587 M/s

Table I. CPU, GPU, and Xeon Phi specifications and random memory access rates. The two numbers for Xeon Phi are memory access rates without and with vectorization.

Phi cores, all cases are bottlenecked by either the hardware I/O capacity or the network line rates.

The rest of paper is organized as follows. In Section II we envision Xeon Phi as a packet processor using micro-benchmarks. Section III discusses and compares offloading schemes and a native acceleration model. Section IV illustrates the design of *Knapp* using Xeon Phi and Intel DPDK. Section V shows macro-benchmarks of our offloading framework and example applications. Section VI discusses related work and we conclude in Section VII.

II. Opportunity of Xeon Phi for packet processing

We begin this section with an overview of the Intel Xeon Phi architecture. Released in November 2012, Intel Xeon Phi is the brand name for the first commercialized generation of Intel's Many Integrated Core (MIC) architecture products, code-named *Knights Corner*. Xeon Phi co-processors assist the main CPUs by adding instruction throughput and high local memory access bandwidth to the system, as shown in Figure 1. Key features of Xeon Phi are as follows:

- The cores are almost compatible with x86 cores, but they come with 512-bits wide vector processing units (VPUs) and four hyperthreads. That is, we can not only run existing pthread-based CPU code without modification using a cross compiler, but also optimize the code by vectorization.
- It offers a wide variety of programming models. We can offload code using conventional models such as OpenCL and OpenMP. In addition, Intel's own language extensions such as Cilk+ and task parallelization libraries such as Intel TBB (Threading Building Blocks) are available. With a native programming model, we can run our own code directly on Phi using pthreads.
- It offers a transparent low-level programming environment, including the source code of its custom Linux and PCIe communication drivers. This allows detailed inspection of all performance characteristics of Xeon Phi and enables development of new offloading schemes, if necessary.

In this work, we use Xeon Phi 5110P with 60 physical cores operating at 1.053 GHz and 8 GB onboard GDDR5 RAM doubling as a file system storage. Each core has 32 KB of L1 instruction and data cache plus 512 KB of L2 cache. It supports four hardware threads (240 hardware threads in total) and executes two instructions concurrently through two pipelines: one optimized for vector processing and the other for scalar-only processing.

A. Packet Processing Workload

Network packet processing requires different computation primitives and offloading mechanism from scientific computations.

In scientific computing, large-size matrix manipulation and floating point operations are the dominant types of computation, and floating point operations per second (FLOPS) is an important performance metric. Network packet processing workloads on the other hand have a very different set of dominant operations: table lookups for address resolution, counter decrements of the Time-To-Live (TTL) field, and pattern matching in intrusion detection, to name a few. These operations are mostly integer operations with conditional branches and memory accesses to I/O DMA buffers, where FLOPS bears little relevance to the packet processing workloads.

In addition to computation power, delay and I/O performance are also important in packet processing. Most packet processing workloads need to satisfy sub-milliseconds of delay constraints, and tens of Gigabytes of data must be processed per second. In the rest of this section we present the key performance metrics that are relevant to network packet processing and compare Phi against CPU and GPU in those metrics.

B. Instruction Throughput

Xeon Phi is an attractive accelerator to support high instruction throughput. Here we compute the theoretical instruction throughput unhindered by memory access patterns. We take the IPv4 address lookup code from a routing information base (RIB) following the DIR-24-8-Basic scheme [10], and examine the theoretical instruction throughput at the assembly level ignoring memory access. We compare the instruction throughput of Xeon Phi against NVIDIA Tesla K10.

The maximum instruction throughput for a Xeon Phi processor running at 1.053 GHz is 1.010 Tops considering 16 integer VPUs. As the assembly code of the IPv4 address lookup algorithm consists of 11 vector instructions, the maximum throughput of IPv4 address lookup on Xeon Phi is 91.8 Glookups/sec assuming Xeon Phi compute vector instructions in a single cycle each.

On the contrary, the assembly code of the same IPv4 address lookup algorithm written in CUDA contains 11 160-IPC (instructions per cycle) instructions, 3 128-IPC instructions, 8 32-IPC instructions, and 3 8-IPC instructions. Taking the clock frequency or 0.745 GHz and 2,560 cores of Tesla

K10 into account, the maximum lookup throughput becomes 16.6 Glookups/sec.

Since the required packet processing rate for our system with four 10 GbE NICs is 4×14.88 Mpps or 59.52 Mpps for 64 B minimum-sized packets arriving at the full line rate, the instruction throughput of both Tesla K10 and Xeon Phi 5110P reaches well beyond it; they can process all packets without drops. The much larger throughput of Xeon Phi indicate its huge potential to do more complicated computations than GPUs. Note that these numbers represent optimistic estimations not accounting for pipeline stalls caused by data dependency and out-of-order memory access.

C. Random Memory Access Rate

A typical modern CPU architecture includes support for out-of-order execution to hide memory access latency. GPUs execute instructions in-order and use hardware-supported context switching for the same purpose exploiting the massive number of threads. Xeon Phi also uses an in-order execution pipeline, but does not support automatic context switching like GPUs. How penalizing is this architectural difference in packet processing?

Address resolution in packets results in lookups on huge tables (a few tens to hundreds of MBs) with random keys (e.g. destination addresses). We used a micro-benchmark devised by Kalia *et al.* [11] to measure the random memory access rate. We report the results of the micro-benchmark in Table I.

Although Intel Xeon Phi has the largest sequential memory bandwidth (320 GB/s), it has the worst random memory access rate among the three. It is even 12.6% worse than CPUs. To compensate for the performance degradation in random memory access, we take advantage of VPUs and vectorize all memory accesses with 512-bit-wide load-and-store and scatter-and-gather instructions. The vectorized micro-benchmark delivers 17.4% improvement and has a better memory access rate than CPUs.

Summary The computation power of Xeon Phi stands on par with latest CPUs and existing accelerators such as GPUs. The architectural details differ but the performance metrics for packet processing such as instruction throughput and random memory access rates are enough to put Xeon Phi as a contending packet accelerator.

III. Why a New Offloading Scheme?

In the previous section we have reviewed architectural features and basic performance data of Xeon Phi that render it as a feasible platform for packet processing. In this section we review existing offloading and computation frameworks available on Xeon Phi such as OpenCL and OpenMP. Unfortunately, we find that they fail to satisfy desired performance requirements in terms of data transfer rates and thread synchronization latency. Nonetheless, the low-level PCIe communication API called SCIF (Symmetric Communication InterFace) and the native POSIX threading model offered by Xeon Phi provide much better performance. This makes Xeon Phi a promising packet processing accelerator but also poses challenges to devise a new offloading scheme tailored for packet processing.

A. Existing Offloading Schemes

The packet processing workload deviates from traditional HPC (High Performance Computing) workloads. Moreover, the overhead of data transfer between the host CPU and the auxiliary accelerator is non-negligible in the overall latency of packet processing. Thus the offloading scheme for the packet processing workload should incur minimum communication overhead, as it is called frequently (e.g., tens of thousands of times per second) for a relatively small amount of data (e.g., 10–100 KBytes).

Table II compares well-known programming models for Xeon Phi. Among them, we choose to evaluate OpenCL and OpenMP. The two are highly popular and are open standards. We exclude Intel Cilk+ and Intel TBB since they do not provide buffer management functions for offloading.

We compare OpenCL and OpenMP against POSIX threads API, where we offload 2,048 IPv4 addresses (8,192 bytes) to DIR–24–8 IPv4 lookup algorithm [10] running on Xeon Phi, and produce a timing breakdown of each offload step. We take a mean latency of each step from 10,000 iterations excluding the first 1,000 for initialization overheads. In all cases, we store lookup tables in persistent memory prior to offloading.

OpenCL: We use a single command queue to copy data from host to device, launch kernel, copy results back from device to host, and synchronize. We use the `clGetEventProfilingInfo` API call for timing.

OpenMP: We use `#pragma offload target` and `#pragma parallel` for compiler directives for data transfer and implicit parallelization. We take the performance figures from threading configurations that show the best offloaded performance. As programming in OpenMP also allows the use of Xeon Phi vectorization intrinsic, we also measure the performance of the same program with manually vectorized co-processor code segment.

POSIX threads API: Our POSIX threads implementation uses `pthread` API for threading primitives, and SCIF API for host-to-device and device-to-host transfers. More specifically, we use SCIF's pull-based DMA transfer primitive (`scif_readfrom()`) for host-to-device transfers, and push-based primitive (`scif_writeto()`) for device-to-host transfers. On Xeon Phi, persistent threads busy-wait on atomic barriers until a single master thread signals worker threads to proceed. Worker threads synchronize again with the master thread upon finishing their share of IPv4 lookup workloads, after which the master thread copies the results back to host. Note that this threading model is simplified for the purpose of evaluating different offloading primitives. We present a more full-fledged threading model in subsection IV-B for *Knapp*, our offloading framework.

Table III shows that each programming model exhibits distinct performance characteristics. Clearly, POSIX threads implementation shows kernel performance close to an order

	Vectorization	Thread Pinning	Thread Sync	Data Copy	Data Sync
OpenCL	△ (hints)	Implicit	OpenCL API	COI daemon	COI daemon
OpenMP	×	Explicit	Implicit	COI daemon	COI daemon
Intel Cilk+	○	Auto	Auto	N/A	N/A
Intel TBB	×	Auto	Auto	N/A	N/A
POSIX Threads	×	Explicit	Explicit	SCIF API	SCIF API

Table II. Comparison of computation schemes available on Xeon Phi.

	OpenCL	OpenMP	POSIX threads
Host → device copy	28	8	6
Kernel execution	110	94	14
Device → host copy	5	51	5

Table III. Timing breakdown of DIR-24-8 lookup offloading (in μs).

of magnitude faster than its OpenCL or OpenMP counterpart, even when OpenMP implementation adopts manual vectorization. OpenCL and OpenMP have higher overhead in kernel execution than POSIX threads. We track down the causes and observe the following. OpenCL exposes two parameters for multithreading: work-group and work-item. Although general guidelines to utilize Xeon Phi with such parameters exist, the exact detail of when a thread spawns and terminates, for instance, remains obscured from the developer. Intel's OpenMP runtime makes use of an additional abstraction layer: the Coprocessor Offload Infrastructure (COI) daemon. It exists to relax implementation complexity of coprocessor buffer management and runs on Xeon Phi as a native user process. Table III illustrates that this comes at a high cost, as every data transfer and synchronization goes through the general-purpose COI daemon process, precluding opportunities for workload-specific optimization.

Summary Existing offloading schemes have significant overheads to be used for packet processing. Now the question is, is this the fundamental limitation of the Xeon Phi hardware or not? While we suspect that those runtimes have their own overheads, we need to confirm the maximum capability of Xeon Phi before concluding.

B. Data Transfer Rate between CPU and Xeon Phi

To sustain a high packet processing rate in the overall system, it is imperative to take full advantage of the maximum data transfer rate between CPU and Xeon Phi. Xeon Phi's PCIe 2.0 16x controller can handle up to 64 Gbps in theory, which is larger than the total 40 Gbps of all traffic from four 10 GbE NICs. Yet, actual data transfer rates vary depending on communication patterns. Therefore we evaluate the DMA transfer costs over a range of payload sizes.

In Xeon Phi's runtime stack, SCIF lies at the bottom. It is the basic building block of all CPU-MIC communication libraries. As CPU always initiates packet processing and does not need to pull from Xeon Phi, we use the push-based call, `scif_writeto()`, to implement the transfer between CPU and Phi. We vary the payload size from 512B to 128KB and execute each size for $1,000$ times.

In Figure 2 we plot the average data rate and latency. DMA exhibits asymmetric transfer rates depending on the direction; host-to-device copy rates are faster up to 1.5×. We suspect that this is due to the difference in clock frequencies of

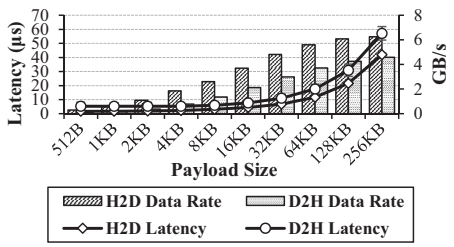


Fig. 2. DMA transfer latency and rate between host and MIC via `scif_writeto()`.

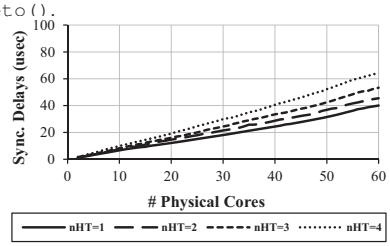


Fig. 3. Average synchronization costs (in μsec) varied by the size of groups (number of cores × number of hyperthreads). Plotted separate lines for different number of hyperthreads using the same axis for easy comparison.

DMA-initiating devices, namely, the CPU and the Xeon Phi processor. Other work has observed the same asymmetry as well [4], [11].

Based on our evaluation, a back-of-envelope calculation shows that the data transfer bandwidth is enough to handle small packets coming at high rates. For example, copying 16 KB of L2 and L3 headers (about 481 packets) is done in 3.7 GB/s, or 57 Mpps using 64-byte packets. This rate means that Xeon Phi can sustain the full line rate of 59.52 Mpps with sufficiently large offload batch sizes (e.g., ~512 packets).

C. Xeon Phi Thread Management and Synchronization

Latency is a critical factor in network routers and middleboxes, and we need worker threads on Xeon Phi to stay responsive to the incoming data from the host. We investigate cache synchronization overheads among Xeon Phi cores to decide how large the thread synchronization group should be and how we should map the threads to physical cores and hyperthreads.

As Xeon Phi's core-local caches and tag directories are on a ring topology, we first check if there is disparity in remote cache access time. If so, choosing an arbitrary pair of cores may have different synchronization costs. We measure the latency for repeating 1 million atomic increments on a variable shared by each possible pair of cores. The latency of all 60 by 60 pairs of cores shows fluctuations, but the pattern changes over repeated executions. As there is no persistent disparity pattern, we assume Xeon Phi cores have uniform access

latency to remote cache and the device memory. That means, we may split the Xeon Phi processor into any combination of physical cores.

Next, we measure how fast the synchronization overhead increases with the number of threads in the group. In our experiment the threads use an atomic barrier and repeat 1,000 times of barrier synchronization. Figure 3 shows the average latency with a different number of hyperthreads (nHT) used per physical core. The latency increases linearly with the number of cores, which begins from under 2 μsec with two cores and peaks at 64 μsec with 60 cores when four hyperthreads per core are used. The latency increase is not proportional to the number of hyperthreads per core. As the barrier-based synchronization occurs at least twice per offloaded packets in batches (subsection IV-B), the amortized cost of atomic barriers is negligible.

Although the synchronization overhead is minimum for four hyperthreads for any number of cores in a thread group, having four hyperthreads on a core may deliver worse performance than running two hyperthreads each on two cores. We defer the choice of the thread group size to subsection IV-B.

Summary We conclude that Xeon Phi hardware is capable of handling multi-10G packet processing workloads, as others have confirmed the same for the GPUs. Built-in offloading APIs have high latency overhead, and we need a new scheme to avoid them. In the next section, we design our new offloading scheme tailored for packet processing.

IV. *Knapp*: Offloading Framework for Packet Processing

In this section, we present our new offloading framework, *Knapp* (*Knights-corner as a Packet Processor*). The goal of *Knapp* is to offload 100% of network packets to Xeon Phi and bring out maximum performance out of Xeon Phi. It covers the complete packet path from NIC to CPU and Xeon Phi then the path out.

A. Host-side of Knapp

Since we offload all packet processing to Xeon Phi, we keep the host-side architecture simple, reusing key performance-improving techniques from previous work [1], [4]. *Knapp* uses Intel DPDK for high-performance packet I/O and multi-core optimized data structures. It also uses the standard RSS (receive-side scaling) feature of 10 GbE NICs to spread incoming packets equally to multiple I/O threads pinned to dedicated individual CPU cores. We leave out one physical core per NUMA socket to run a dedicated thread for device communication. This device thread polls for offloaded task completion and pushes finished tasks to corresponding I/O threads. Previous work has chosen this design to avoid device API runtime overheads, and our micro-benchmark also confirms that the same holds for Xeon Phi; accessing Xeon Phi from multiple worker threads degrades the overall forwarding throughput by ~2 Gbps under minimum-sized packet workloads.

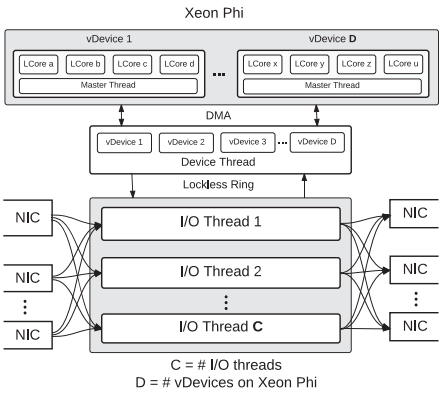


Fig. 4. Architecture for offloading packet processing on Xeon Phi.

B. Resource Partitioning with `vDevice`

To avoid unexpected performance degradation due to interaction between many cores, we need to carefully partition the processor resources. For example, synchronizing all 60 cores $\times$ 4 hyperthreads/core at once is costly. As we have seen in Figure 3, the synchronization delay alone could amount to 60 μsec. In such a case the overhead of synchronization overshadows the time spent for computation and decreases Xeon Phi utilization for packet processing.

Instead, *Knapp* partitions the cores into groups or `vDevices`. A `vDevice` is a set of logically consecutive and physically adjacent cores that runs a thread group progressing together. Individual `vDevices` are associated with a packet processing application and two SCIF channels: one for receiving synchronous control messages (control) and the other for packet offloading (data). The control channel treats relatively small 32-byte control messages comprised of opcodes and a few parameters through memory-mapped I/O (`scif_send()` and `scif_recv()`), and the data channel operates multi-KB DMA transfers (`scif_writeto()`) and sends asynchronous signals (`scif_fence_signal()`) as completion notification to the remote SCIF peer. Different `vDevices` processing the same type of workload share common data structures such as IP routing tables via read-only memory. This read-only memory utilizes Xeon Phi's globally coherent L2 cache and saves memory bandwidth.

C. Packet Processing Workflow

Figure 4 shows the software architecture of *Knapp*, highlighting the interactions among multiple packet I/O threads on the host and a device handler thread that controls Xeon Phi, as well as `vDevices` performing computation inside Xeon Phi.

Along with the host and Xeon Phi thread hierarchy, the sequence of the whole packet processing workflow is as follows. On the host-side, each I/O thread receives and stacks up to B packets in a single batch, where B is the offload batch size. When the stack becomes full, the worker thread extracts and serializes regions of interest (RoI) for every packet and copies them to the host-side offload buffer. The RoI is a manually defined range that will be accessed by the offloaded application. The host-side device thread monitors the offload

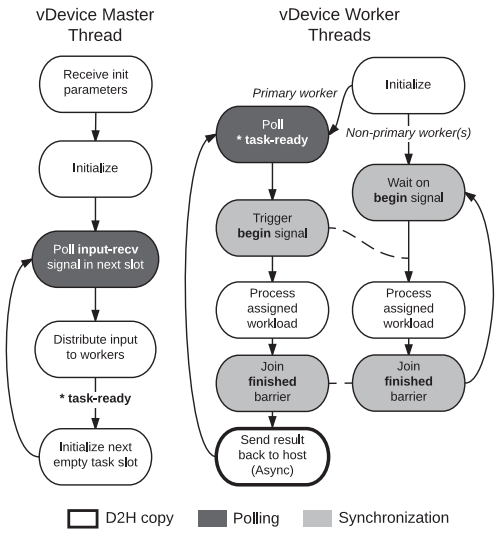


Fig. 5. Control flow diagram of `vDevice` pipeline.

buffer and asynchronously pushes the serialized RoI payloads via the data channel to a `vDevice`.

Figure 5 illustrates what happens next on the Xeon Phi side. Each `vDevice` has a *master* thread that manages synchronization with the host and one or more *worker* threads that performs actual packet processing together. The master thread handles control messages and polls the next task slot that is written by the CPU (**input-recv**). If the input is available, it distributes the data to workers and then marks the next task to be ready (***task-ready**). The primary worker thread polls until the next task is marked ready, triggers the **begin** signal for other worker threads to start processing their share of the input, synchronizes again with worker threads using the **finished** barrier, and finally sends the processed result back to the host memory.

Meanwhile, the host-side device thread polls the in-flight task queues of all `vDevices` simultaneously for completion, and sends back finished tasks via lockless completion queues to their originating host I/O threads. In order to overlap data transfers and computation and increase throughput as in [4], [12], [1], we make each step to use `vDevice` to run in parallel though for a single `vDevice` those steps run in a sequential manner. We pool N input buffers, output buffers, and signal slots, where N is the configurable depth of the processing pipeline.

D. Knapp Applications

We demonstrate *Knapp* as a full-suite packet processing system by running simplified IPv4 and IPv6 router applications. Our implementation focuses on route lookup performance and not on routing protocols. The RoI of our routing applications include L2 (Ethernet) and L3 (IP) headers.

IPv4 Router: We use the DIR-24-8 lookup algorithm [10] with 282K routing prefixes matched over packets with randomized IP destination addresses.

We have built two versions of IPv4 router: scalar and vectorized. The scalar version is a direct porting of the CPU-based implementation from NBA [1]. We simply recompiled it with the cross compiler. The vectorized version takes 16 destination addresses at once and performs table lookups in parallel. Manually vectorizing the DIR-24-8 algorithm is straightforward, as it is free from per-iteration function calls, data dependency between iterations, and complex conditional branches that complicates vectorization. Every conditional branch in our vectorized DIR-24-8 simply eliminates packets with invalid fields from vector masks as it progresses, setting them aside from further computation. As the two memory lookups used in DIR-24-8 range over a large address space (64 MB), the vectorized version benefits from the scatter-and-gather memory access feature of Xeon Phi by coalescing at most 16 memory accesses in a single instruction, whose performance gains is demonstrated in Table I.

Category	Specification
CPU	1x Intel Xeon E5-2670 (Sandy Bridge, octa-core 2.6 GHz, 20 MB L3 cache)
RAM	32 GB (DDR3 1,600 MHz 4GB x8)
NIC	2x Intel 82599ES (dual-port 10 GbE, total 40 Gbps)
MIC	1x Intel Xeon Phi 5110P (60 1.053 GHz Atom cores, 8 GB on–board RAM, 320 GB/s, PCIe 2.0)

Table IV. Hardware configuration

IPv6 Router: We use a binary-search-based IPv6 address lookup algorithm suggested in [13]. As with the IPv4 router, we have begun from porting the CPU-based scalar implementation from NBA [1] and then written the vectorized version.

However, vectorizing the IPv6 lookup algorithm is a lot more challenging than the DIR-24-8 algorithm. First, it uses a two-level hash table that complicates memory accesses. Although Xeon Phi supports scattered loads and stores upon a single base address with multiple offsets, we cannot make scattered loads with multiple different base addresses. Thus we have to flatten the two-level hash table into a single layered hash table, which requires additional address calculations on each lookup. Second, the control flow diverges during the binary search and hash bucket chaining. Xeon Phi supports instruction masking to select specific field of a given vector. Yet, masked instructions do not eliminate suffering from unrolling every possible control flow into the masks.

We leave automatic vectorization of existing scalar code using rich vector instruction sets with the masked execution model for future work.

V. Evaluation

A. Experiment Setup

Our setup uses a single commodity server running Ubuntu Linux 14.04 with generic Linux kernel 3.13.0–46, Intel DPDK 2.2 development version, and Intel MPSS 3.4.2. Table IV lists the system information for reference. For all benchmarks, we use a DPDK-based packet generator running on a separate server connected via a 10G switch. The generator sends up to 40 Gbps of synthetic IPv4/v6 traffic in varying packet sizes ranging from 64 bytes to 1,500 bytes of the maximum

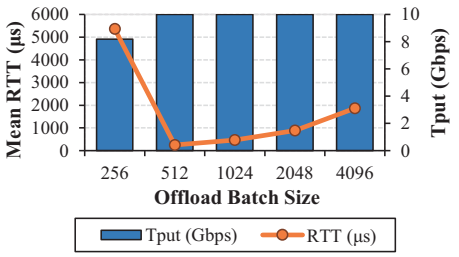
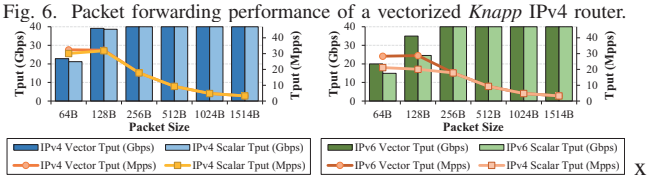


Fig. 6. Packet forwarding performance of a vectorized *Knapp* IPv4 router.



(a) Vectorized/Scalar IPv4 routing (b) Vectorized/Scalar IPv6 routing

Fig. 7. Packet forwarding throughput for IPv4/v6 routing workloads under varying inbound packet sizes ($B = 512$). Each `vDevice` makes use of 8 contiguous physical cores each running 2 hyperthreads.

transmission unit. We configure *Knapp* to use only two hyper-threads out of four available in each physical core of Xeon Phi, as our experience shows that running more threads degrades the performance and Phi cores can run two instructions in parallel.

B. Offload Batch Size

The offload batch size is the key performance parameter that trades per-packet forwarding latency for offloaded parallelism and forwarding throughput. For each routing application, we vary the offload batch size B (packets per each offloading operation) from 256 to 4096, and measure the round-trip latency (RTT) along with the forwarding throughput.

We limit the ingress traffic to 10 Gbps and distribute it over four 10 GbE cards to ensure that there is no side-effect caused by NICs' queuing delays.

Figure 6 shows the results. The processing latency of *Knapp* grows linearly as B increases as expected, except when $B = 256$. The reason for exceptionally high latency when $B = 256$ is the excessive polling overhead. We find that the offload batch size $B = 512$ that has the lowest latency (249 μsec) is consistent with our back-of-envelope estimation from subsection III-B. We use this batch size as default for later experiments.

C. Packet Sizes and Vectorization

Next, another important factor for *Knapp*'s performance is the packet size. Though there will be a wide range of packet sizes mixed in real traffic, we focus on the performance impact by the fixed packet sizes in our evaluation. In this scenario, computation cost is fixed among packets and I/O performance requirement will vary. We also compare the scalar and vectorized versions of our applications to show the vectorization effects of Xeon Phi.

In this experiment, we offer full rates of ingress traffic, 40 Gbps in total, and measure the throughput in both Mpps and Gbps to highlight the cause of performance bottlenecks.

Figure 7 plots the forwarding throughput with different inbound packet sizes. First, it shows that the major bottleneck is I/O rather than computation. For 64-byte and 128-byte packets in both vectorized IPv4 and IPv6 applications, the throughput in Mpps reaches the I/O bound (the flat part) and shows Gbps less than line rates while they achieve line rates for larger packets. Note that previous work also has pointed out this PCIe I/O bound; the PCIe bus utilization drops below theoretical lane bandwidth when minimum sized packets flood the NICs [4], [14]. This means that a large number of packets incur more transaction overheads inside the PCIe and memory bus despite our packet I/O batching and offload batching.

Second, it shows that vectorization is essential to exploit the full computation capacity, particularly for heavy computations. The difference between scalar and vectorized versions is more pronounced in IPv6 routing than IPv4 routing, as shown in Figure 7(b) and 7(a). In IPv6 with 128-byte packets, the vectorized version gains 42.6% more throughput compared to its scalar equivalent. In contrast, both scalar and vectorized versions of IPv4 show little difference in throughput. This states that the computation is not the bottleneck but the I/O in the case of IPv4.

D. `vDevice` Group Size

As we partition the cores of Xeon Phi into multiple groups (`vDevice`), the number of cores in each `vDevice` may cause performance variation due to differences in internal synchronization and cache sharing overheads.

To evaluate that, we fix the number of I/O threads and the number of `vDevice` groups to 7, since our setup has 7 I/O threads in the host and thus the maximum number of `vDevice` groups is 7. Given 7 groups and 60 cores in the Xeon Phi processor we use, the available range of the number of cores per `vDevice` is 1 to 8. For simplicity, we only consider equal-sized `vDevice` groups.

Figure 8 illustrates the throughput variation under the full load, with a varying number of cores per `vDevice` group. It shows the benefits of scaling out a `vDevice` from one to eight physical cores: the forwarding throughput increases 3.9 to 19.98 Gbps, $\times 5$ increase, and the per-batch processing latency (except packet I/O) improves from 592 down to 79 μs, an $\times 7.4$ improvement. This result tells that the internal synchronization and cache sharing overheads among Xeon Phi cores are negligible and `vDevice` groups should have as many cores as they can. In prior experiments, we have used 8 cores per `vDevice` groups.

Summary With a set of evaluations above, we find that our new offloading scheme *Knapp* successfully achieves both high throughput and low latency comparable to existing GPU-based packet processing accelerators. We believe that the future generations of Xeon Phi that will use faster PCIe standards and/or offer integrated network controllers could boost the performance even further.

VI. Related Work

Hardware-assisted packet processing: Network packets, not considering ordering constraints, are independent data

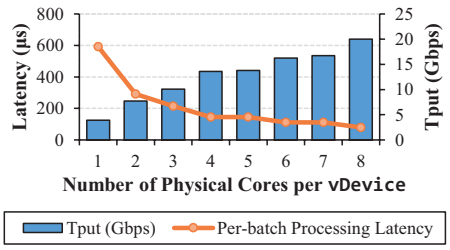


Fig. 8. Forwarding throughput and per-batch processing latency of vectorized IPv6 router. Each physical core runs 2 hyperthreads.

segments going through homogeneous processing pipelines, which makes them suitable targets for parallel computation.

Along with general purpose CPUs, using accelerators help to achieve high performance. GPUs are typically used to boost computation power [15], [5], [4], [1], [16], [17], [18], [12]. GASPP[7] demonstrates a purely GPU-based packet processing using a hybrid offload scheme that switches between GPU-NIC zero-copy and copying buffers of back-to-back network packets depending on the individual sizes of packets received.

Applications exposing performance characteristics of Xeon Phi: MRPhi[19] implements Xeon Phi-specific optimizations on top of Phoenix++[20], a MapReduce[21] implementation targeted for multicore, shared memory systems. Popcorn[22] eliminates the need for re-programming applications to run on Linux OS-capable heterogeneous platforms (i.e. Xeon Phis) through the use of a compiler framework capable of producing executables optimized for all heterogeneous ISA platforms.

VII. Conclusions

We design and implement an efficient computation offloading framework for Xeon Phi, *Knapp*, to support low latency and streamed data for packet processing workload. We have devised `vDevice`, a virtually partitioned view of Xeon Phi cores to balance parallelism and synchronization overheads. The evaluation results using simplified IPv4 and IPv6 routing applications show that the performance of our Xeon Phi-based packet processor reaches the line rate of 40 Gbps on a single commodity x86 server.

The question of Intel Xeon Phi as a packet processing accelerator is yet to be resolved. Remaining evaluations are: extensive evaluation of network applications on Knapp; performance comparison of those applications against CPUs and GPUs; incorporation of new features of the updated Intel Xeon Phi architecture. Recent push in microprocessor architecture comes from data mining and deep learning. The verdict is yet out on what impact the recent architectural changes have on packet processing.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014007580). We thank Jaimie Park and Shinae Woo for review on the structure of the paper and anonymous reviewers for their precious feedbacks.

References

[1] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors," in *Proc. of the tenth ECCS*. ACM, 2015, p. 22.

[2] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing," in *Proc. of the eleventh ACM/IEEE ANCS*. IEEE Computer Society, 2015, pp. 5–16.

[3] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, "The power of batching in the click modular router," in *Proc. of the APSys*. ACM, 2012, p. 14.

[4] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195–206, 2011.

[5] W. Sun and R. Ricci, "Fast and flexible: Parallel packet processing with GPUs and Click," in *Proc. of the ninth ACM/IEEE ANCS*. IEEE Press, 2013, pp. 25–36.

[6] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O." in *Proc. of the USENIX Annual Technical Conference*, 2012, pp. 101–112.

[7] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: a GPU-accelerated stateful packet processing framework," in *USENIX ATC*, 2014.

[8] "*TOP500* supercomputer sites," http://www.top500.org/lists/2016/11/.

[9] R. Rahman, *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[10] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *IEEE INFOCOM*, 1998.

[11] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the Bar for Using GPUs in Software Packet Processing," 2015.

[12] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors." in *NSDI*, 2011.

[13] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, *Scalable high speed IP routing lookups*. ACM, 1997, vol. 27, no. 4.

[14] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, High Performance Ethernet Forwarding with CuckooWwitch," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 97–108.

[15] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster, "Switchblade: a platform for rapid deployment of network protocols on programmable hardware," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 183–194, 2011.

[16] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *Proc. of the 2012 ACM CCS*. ACM, 2012, pp. 317–328.

[17] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: a multi-parallel intrusion detection architecture," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 297–308.

[18] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 116–134.

[19] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh, "Optimizing the mapreduce framework on intel xeon phi coprocessor," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 125–130.

[20] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: modular MapReduce for shared-memory systems," in *Proceedings of the second international workshop on MapReduce and its applications*. ACM, 2011, pp. 9–16.

[21] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[22] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: bridging the programmability gap in heterogeneous-ISA platforms," in *Proc. of the Tenth ECCS*. ACM, 2015, p. 29.