Exploiting Integrated GPUs for Network Packet Processing Workloads

Janet Tseng, Ren Wang, James Tsai, Saikrishna Edupuganti, Alexander W. Min
Shinae Woo†, Stephen Junkins‡ and Tsung-Yuan Charlie Tai
Intel Labs & Intel Visual and Parallel Computing Group‡
Department of Computer Science, KAIST†
Email: {janet.tseng, ren.wang, james.tsai, saikrishna.edupuganti, alexander.w.min, stephen.junkins, charlie.tai}@intel.com
shinae@an.kaist.ac.kr

***Abstract*—Software-based network packet processing on standard high volume servers promises better flexibility, manageability and scalability, thus gaining tremendous momentum in recent years. Numerous research efforts have focused on boosting packet processing performance by offloading to discrete graphics processing units (GPUs). While integrated GPUs, residing on the same die with the CPU, offer many advanced features such as on-chip interconnect CPU-GPU communication, and shared physical/virtual memory, their applicability for packet processing workloads has not been fully understood and exploited. In this paper, we conduct in-depth profiling and analysis to understand the integrated GPU's capabilities, and performance potential for packet processing workloads. Based on that understanding, we introduce a GPU accelerated network packet processing framework that fully utilizes integrated GPU's massive parallel processing capability without the need for large numbers of packet batching, which might cause a significant processing delay. We implemented the proposed framework and evaluated the performance with several common, light-weight packet processing workloads on the Intel® Xeon® Processor E3-1200 v4 product family (code-name Broadwell) with an integrated GT3e GPU. The results show that our GPU accelerated packet processing framework improved the throughput performance by 2–2.5x, compared to optimized CPU-only for packet processing.**

I. Introduction

Recent years have witnessed the fast deployment of software-defined networking (SDN) and Network Functions Virtualization (NFV) in data center environments and telecommunication providers. The trend propels the need for high-speed software-based packet processing on standard high volume servers. However, the recent surge of network I/O bandwidth, with 100 Gbps Ethernet coming to market, has put pressure on CPUs, caches, and the system memories of multicore servers to sustain both packet processing and NFV services [1].

GPUs have emerged as a promising candidate for offloading network packet processing workloads from the CPU in order to achieve higher full-system performance and freeing more CPU cycles for other application services, e.g., packet forwarding [2], [3], [4], [5], Secure Sockets Layer (SSL) encryption/decryption [6], and regular expression matching in intrusion detection systems [7]. However, most of existing approaches have focused on designing offloading architecture for discrete GPUs. Studies have shown [5], [8] that for discrete GPUs, large batches of packets need to be sent to GPU units to amortize the high CPU-GPU communication latency via the PCIe* (PCI Express*) bus. This could impact the packet processing latency significantly, especially for high-speed data center or carrier networks where such extra latency is not desirable. Meanwhile, proper system optimization for packet processing, such as group-prefetching and software pipelining, on the CPU side has proven to be effective in improving CPU performance [8].

Recently, integrated GPUs—where the CPU and GPU are located on the same die and communicate through on-chip interconnect instead of PCIe—have become more popular in modern server architectures. Integrated GPUs often offer advantages, over the conventional discrete GPUs, of reducing total system power and cost. The additional computational resources provided by such integrated GPUs could be beneficial for network packet processing workloads.

This paper aims to fully exploit using integrated GPUs for offloading packet processing workloads in modern server architectures. To this purpose, we first designed several micro-benchmarks to profile and understand the integrated GPU capabilities regarding packet processing workloads, for example, computation capabilities, communication latency, and random memory access speed. Armed with that insight, we propose a new network packet processing architecture to take advantage of both CPU and integrated GPUs to achieve better system performance. Our proposed framework is carefully designed to incorporate several specific features that are feasible only with integrated GPUs, including (i) Continuous Threads to eliminate the need for launching kernel for every batch of packets, and (ii) a multi-buffering technique that further hides communication latency without requiring large packet batches.

We implement the proposed architecture on the platform based on the Intel® Xeon® Processor E3-1200 v4 product family. We then characterize and evaluate the performance of Intel's integrated GPUs for several network packet processing workloads. We focus on evaluating packet processing applications with relatively light computation tasks, aiming to answer the fundamental question of *whether using integrated GPUs is beneficial for common, lightweight packet processing*; it is well understood that heavier computational workloads will benefit from the GPU's massive parallel computation power more [8]. Our experiments show that our carefully designed and optimized CPU-GPU packet processing framework can effectively improve the CPU only throughput performance by 2–2.5x, without batching large number of packets.

II. Motivation and Background

Network packet processing workload is inherently highly parallelizable at the packet or flow level. Therefore, GPUs with multiple execution units can be a good alternative compute resource for such workloads, with packets being effectively distributed into hundreds or thousands of cores. Batching

TABLE I
INTEGRATED AND DISCRETE GPU COMPARISON.

	Low-/mid-end discrete GPU	Integrated GPU
Power	High (150∼300 W for GPU only)	Low (∼65 W for the whole package)
Cost	$130–$150 for GPU	Comparable to low-end GPU
Communication channel	PCIe* (∼300 cycles)	On-chip interconnect (less than 100 cycles)
Memory model	Large local memory	Large shared system memory with CPU
Total kernel launch latency	High (∼50 μs)	One-time cost in our framework

is important to fully utilize GPU cores and amortize the GPU kernel management overhead. Batching thousands of packets [5], however, also increases processing latency significantly, which is an important performance metric, especially for many real-time applications and for core network packet forwarding processing. Achieving both high performance and low latency is a challenging problem for discrete GPUs.

Integrated GPU offers several benefits. First, compared to discrete GPUs which communicate with CPU cores through off-chip PCIe buses, integrated GPUs provide an on-chip communication channel with much lower latency. Second, this architecture design also facilitates Shared Physical Memory (SPM) and Shared Virtual Memory (SVM) support. SPM/SVM enables the GPU and CPU to share the memory space and same virtual address space, without having to copy data back and forth in the case of discrete GPUs. The SVM provides architecture support for CPU-GPU coherency. To enable programmers to easily take advantage of this feature, carefully designed programming model needs to be used for that purpose. We used OpenCL 2.0 [9] in our framework.

Table I summarizes the high-level comparisons between integrated GPUs and low- to mid-end discrete GPUs that achieve similar computational performance. In general, integrated GPUs provide a low cost and low power alternative over traditional discrete GPUs.

III. UNDERSTANDING THE CAPABILITIES OF INTEGRATED GPUS FOR PACKET PROCESSING

Packet processing workloads inherently have the following characteristics: (i) Highly parallelizable at flow/packet level. (ii) Very memory intensive, usually involving frequent data loading for packets and also for lookup on related data structures. Thus, communication latency between the host and device is critical. (iii) Very little data locality for most packet processing workloads.

We designed and tested three micro-benchmarks, corresponding to the three characteristics of packet processing workloads mentioned above, to understand integrated GPUs' strengths and weakness. The three benchmarks focused on parallel computation, CPU-GPU communication latency, and random memory access performance, respectively. All measurements were performed on servers based on the Intel® Xeon® Processor E3-1200 v4 product family with integrated GT3e. Table II lists the specifications of the server.

A. Integrated GPU's Strength for Parallel Computing

The architecture of EUs on Intel® Processor Graphics is a combination of simultaneous multi-threading (SMT) and fine-grained interleaved multi-threading (IMT) which enables efficient SIMD parallelism, resulting in very efficient parallel computation. Software network packet processing workloads often involve hash computation and lookup on a portion of headers or even payload to perform classification for forwarding, accounting, and security purposes. In order to evaluate the benefit of integrated GPUs that can potentially provide over CPU on computation perspective, we performed hash calculations on CPU and GPU respectively.

TABLE II
SPECIFICATION FOR SERVER BASED ON THE INTEL® XEON® PROCESSOR E3-1200 V4 PRODUCT FAMILY (CODE-NAME BROADWELL).

CPU	4x Intel® Xeon® Processor E3-1285L v4 @3.4 GHz
GPU	GT3e @1.5 GHz
GPU specific L3$	768 KB
Cache	6 MB Last level cache
DRAM	32 GB DDR3 @1.5v

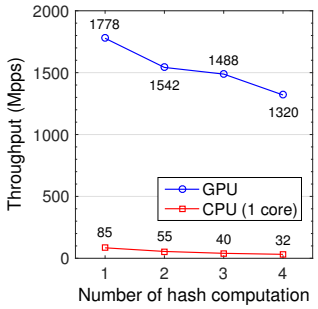

Fig. 1. Comparison of CPU and GPU computation power.

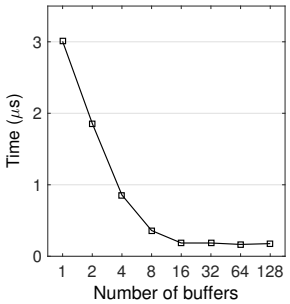

Fig. 2. Execution time (32 packets per buffer) for CPU-GPU communication.

The throughput results are shown in Fig. 1. We can see that due to the GPU's massive parallel computing capability, the GPU outperformed one CPU core by at least one order of magnitude. With relatively heavy computing tasks (4 hash computations), GPU shines the brightest, achieving 40x of single core CPU throughput, measured in Mpps (Million packets per second).

B. CPU-GPU Communication Latency

A typical GPU working thread relies on the host (CPU) to notify the GPU device when the producer data is ready to be processed. As measured in previous works [5], [8], using discrete GPUs for packet processing often suffers significant delays from thousands of kernel launching and communication overhead per second. The non-negligible kernel launch latency (∼4.6 to 9 clocks per thread) motivated us to explore a new programming model that is only realistically feasible with integrated GPUs and SVM support—Continuous Thread. With Continuous Thread, kernel threads only need to be launched once and continuously running after. We explain the details of Continuous Thread in Section IV-B.

For data communication latency, we first conducted an experiment on a simple CPU-GPU latency test. The test starts with the main CPU thread notifying the GPU that input data is ready. As soon as the GPU receives the notification, the GPU loads the data items and writes back a one byte integer and notifies the CPU read thread that the data is ready to be read back. The CPU read thread consumes the output buffer

and releases it back to the main thread when the empty buffer is reused. The timer starts right before the main CPU thread notifying the GPU and stops right after the empty buffer is released by the read thread. This process accurately measures the CPU-GPU communication latency. We set each input data item to be 24 bytes (comparable to IP header lookups). Each batch of communication consisted of 32 input items, which is the same as the DPDK environments where network I/O processes packets in 32-packet batches. The initial result of a single batch transmission revealed that the communication latency was around 3 μs per batch. While this is $\sim$30 % less than the discrete GPU communication latency [8], this is still relatively high for packet processing workloads.

A typical solution to combat the overall latency is to have a large batch to amortize the latency. With integrated GPUs, our target is to minimize the latency and yet fully utilize the GPU's compute power without buffering large batch of packets. With Continuous Thread where all threads are running continuously and polling incoming data, instead of shipping data over one single large buffer, we introduced a "multi-buffering" processing model to hide the latency. By increasing the number of buffers needed to be processed, as soon as the CPU main thread prepares the first buffer and releases it to GPU, it will continue to prepare the next buffer and push it into the pipeline while waiting for the previous buffer to come back. The timer starts before the first buffer is sent to the GPU and stops at the very end after numbers of iterations. The average execution time is calculated as total time, *t*, divided by total number of iterations, *iter*. We explain the multi-buffering technique in detail in Section IV-C. Fig. 2 shows the result of the average execution time for communication of each batch with a different number of buffers. By increasing the number of multi-buffer chains, we could hide the latency by 17x, and the latency converges when it goes beyond 16 buffers.

To understand how fast integrated GPUs can access random data structures, we used 10,240 work-items to ensure that the GPU was fully utilized. Each work-item accessed a 16-byte location in a random fashion within a fix-size large buffer. The random memory access rate *R* is the total number of accesses divided by the total time consumed to complete all accesses. To guarantee that the memory access pattern is random, the fix-size buffer *B* consisted of a list of many 16-byte nodes, which were randomly linked together without a particular order. Each work-item started from a random node inside the list and then walked down 250 nodes. The experiment was run for 50 iterations which resulted in 128 M random memory accesses.

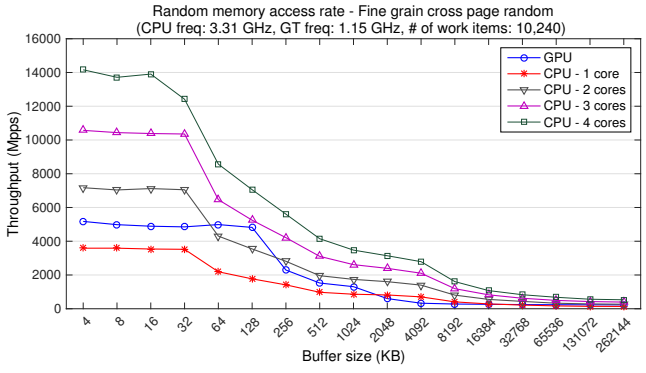

Fig. 3. Random memory access rate.

As shown in Fig. 3, we compared integrated GPU memory access rate with a different number of CPU cores running the same program. The integrated GPU outperformed one core CPU and was surpassed by the multicore CPU in most cases. The reason is that the current GPU is connected to the on-chip interconnect by one data port, and this port becomes a source of contention when access rate increase. This observation suggests that when using GPUs for packet processing, it is desirable to design the algorithm and framework to limit the random access into the system memory. A good example is Cuckoo Hashing [10], [11] which improves hash table memory efficiency significantly. In this paper, we also use Cuckoo Hashing to optimize both CPU only and GPU accelerated packet processing.

IV. Packet Processing Framework with Integrated GPUs

The high-level packet execution flow remains similar to many existing GPU-based packet processing work such as PacketShader [5] where CPU cores perform network I/O tasks and pass the data to GPU for lookup. We introduced two new features in this study that help reducing latency significantly: (i) Continuous Thread which eliminates the need to launch a GPU kernel thread for every batch, and (ii) a multi-buffering technique which allows efficient latency hiding to maximize GPU utilization.

A. Network I/O

For the GPU accelerated packet processing framework design, we used the Intel® Date Plane Development Kit (DPDK) [12] to directly access the NIC from user space, bypassing kernel stack overheads. The DPDK provides a set of open source libraries and user-mode NIC drivers to enable high-speed packet processing on general-purpose Intel® x86 architecture machines. Note that we used the DPDK for both CPU/GPU integrated packet processing and CPU-only packet processing, for a fair comparison.

B. Continuous Thread

Continuous Thread enables launching GPU threads one time and continuously processing incoming packets until an application initiates a termination call. Continuous Thread differs from "persistent thread" used in discrete GPU [13], [14] in the sense that the persistent thread processes work items from a global work pool and terminates itself once the work pool is exhausted. This model is similar to the previously mentioned "batching" technique, but prevents the GPU from starvation via software scheduling. However, this method does not address the latency problem for typical latency-sensitive producer-consumer applications, such as network packet processing, due to the lack of instant interaction between the CPU and GPU. On the other hand, the proposed Continuous Thread can process the continuous streaming data without batching a large work pool, hence improving the latency performance. In our model, the working thread continuously processes new stream-in data until the application initiates a termination call, making it truly persistent. Similarly, the host application (CPU) could immediately process the GPU output without waiting for the remaining work in the pool to be finished.

In our system design, we utilized a shared control object to synchronize communication between the GPU and CPU. The

update of this control object is instantly visible by the GPU and CPU due to the CPU/GPU coherency support introduced by Intel® Processor Graphics Gen8. GPU threads spin-wait for CPU flags to be updated after the next batch of packets is ready to be processed, while the CPU thread spin-waits for GPU flags to be updated after the lookup operation is finished.

Continuous Thread requires only a one-time cost of kernel thread launching and command setup and lets the kernel threads continuously poll the incoming new data to process. With this, our framework successfully removed the thread launching and software setup overhead.

C. Multi-buffering Technique

With multiple buffers, the CPU as the producer continuously replenishes the buffers that are ready to be used while the GPU as the consumer is processing the newly filled buffer. Given that the latency hiding effect converges with the increasing number of multi-buffers, we fixed the number of buffers at 72 and varied the number of data items (packets) needed to be loaded and written back within each buffer. As the number of packets per batch increases, the number of corresponding work-items also increases, which might increase the GPU utilization. We ran experiments with batch size varying from 1 to 64, and results show that the CPU-GPU communication overhead is fairly consistent with the batch size, hence a large batch size might not be necessary for optimal performance. Fig. 4 shows the proposed GPU accelerated framework with Continuous Thread and the multi-buffering technique.

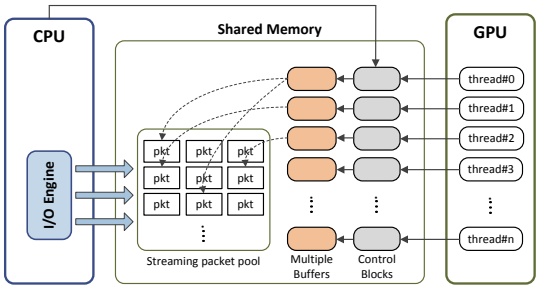

Fig. 4. GPU accelerated packet processing framework.

V. Performance Evaluation

A. CPU Packet Processing Optimization

For a fair comparison, we used an optimized CPU packet processing module, which is also described in [12], [8], [11]. Features include:

Highly vectorized packet processing. DPDK takes advantage of Intel® Advanced Vector Extensions (Intel® AVX) instructions [12] to minimize processing cycles and optimize packet processing performance. In this work, we employed the same methods used in DPDK.

Software pipelining and group prefetching. Software pipelining processes a batch of packets in several stages; the stages are carefully designed so that group pre-fetching can be issued to hide the memory access latency. The concept is similar to hardware pipelining to fully utilize the CPU resources. By intelligently scheduling computing "stages" after data prefetching, useful work can be done while waiting for memory access [12], [8].

Cuckoo Hashing for fast table lookup. Cuckoo Hashing is used to improve the hash table lookup performance for L2 switching and IPv4/IPv6 flow classification. Cuckoo Hashing achieves high memory efficiency of hash tables while ensuring expected $O(1)$ retrieval time. Studies [8], [11] have shown that using Cuckoo Hashing can improve lookup performance significantly.

B. Workload Description

Layer 2 switching. We used a hash table to look up the destination MAC address of the packet as the key to determine the egress port. We used the Cuckoo open addressing hashing scheme [11], which requires, on average, 1.5 hash computations and 1.5 memory accesses per lookup.

IPv4/IPv6 flow classification. We evaluated flow classification performance of both IPv4 and IPv6 based on hash table lookup. Different sizes of the flow table were evaluated to understand the impact of table size on performance.

IPv4 forwarding. We used the DPDK's Longest Prefix Match (LPM) implementation that uses the DIR-24-8-BASIC algorithm [15] for IPv4 destination IP address lookups. In our tests, we populated the table with 1,076,806 prefixes using the DPDK test app, which follows a real-world route table distribution [16]. Since the algorithm is essentially an indexing scheme, the computation is negligible and on average it requires $\tilde{1}$ memory access per lookup.

IPv6 forwarding. We used the DPDK's LPM implementation for IPv6 destination IP address lookups. The distribution of prefix lengths used in our evaluation is included in the DPDK's test application. Packets with IPv6 address listed in `large_ips_table` [17] were used for the lookup, with prefix length varying for 5 to 128 bits. IPv6 forwarding required more memory accesses (4–6 accesses) than IPv4 forwarding due to the longer address.

C. Experiment Setup

The hardware configuration is the same as described in Section III. For software configuration, since we have 48 EUs to work with, we used 144 multi-buffers to work with Continuous Thread, which means there will be 4,608 packets being processed simultaneously during peak time with a batch of 32. It is worth noting that for CPU-only processing, we used a run-to-completion model which is the most efficient implementation in terms of achieved throughput; with this model, CPU-only throughput roughly scales with the number of cores.

D. Workload Generation

We needed to rely on the shared memory coherency of Intel® Processor Graphics Gen8 to implement the Continuous thread feature. However, the current OpenCL 2.0 driver has very limited support on the Linux* platform, which for now limited our choice of implementation to the Microsoft Windows* OS, while the newest version of the DPDK is optimized for Linux. This makes it difficult to incorporate DPDK I/O in our GPU accelerated processing implementation. On the other hand, with the evolution of the pipelined packet framework, I/O becomes a fix cost on the receiving CPU core in the equation. Hence, we emulated the real-time I/O via a random packet generator that generates packets with actual packet format and adds I/O processing cost (∼47 cycles per packet as measured on real system) before handing them for processing in GPUs. Our emulated I/O produces the same

I/O performance as DPDK implementation on real system (∼60 Mpps).

To generate random packets, we first created a 1 GB huge buffer that contains 16 M of 64-byte packets. We then dedicated a 32-packet-size queue for each multi-buffer. During runtime, for each process iteration, the CPU core that is responsible for I/O randomly appointed a starting address to each queue for GPU to take 32 packets starting from that address

E. Throughput Comparison and Analysis

With our integrated GPU accelerated packet processing framework, we no longer need a dedicated master thread for CPU-GPU communication as in the case of discrete GPUs [5], [8]. In our evaluation, for GPU packet processing, we used a single CPU core to perform network I/O tasks and also communicate with the GPU; while for CPU-only processing, we used a single core with a run-to-completion model, which is the most efficient CPU processing model for lightweight packet processing workloads. *The multi-core lookup performance scales roughly linearly with the number of cores, until the throughput reaches the platform hardware limitation.* In the case when one CPU core cannot saturate the GPU processing power, such as IPv4 and IPv6 forwarding, we used multiple CPU cores for I/O tasks. All experiments are run with batch size of 32.

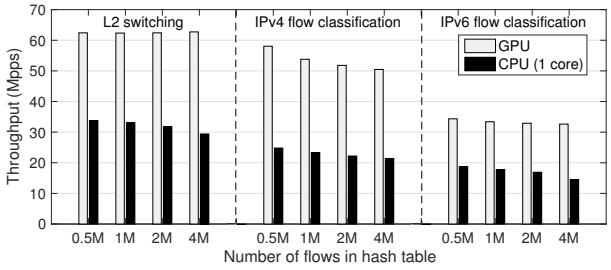

Fig. 5. Flow classification performance comparison.

Fig. 5 shows the throughput comparison for L2 switching and IPv4/IPv6 flow classification. For all cases, enabling the GPU for packet processing provides more than double the performance of a single CPU core and also scales better with the number of flows. To illustrate the performance comparison between integrated and discrete GPU, for IPv6 classification, Packetshader [5] using GTX480 achieves ∼1.64 Mpps with batches of 64 packets while the integrated GPU achieves much higher throughput at ∼15 Mpps with smaller batch size of 32.

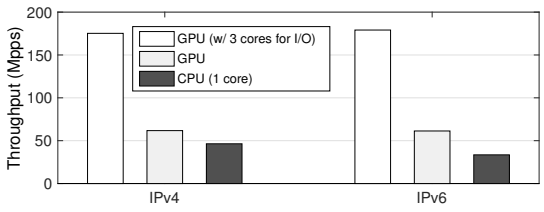

Fig. 6. IPv4/IPv6 forwarding performance comparison.

The IPv4 and IPv6 forwarding results are shown in Fig. 6. First we observed that enabling the GPU only provided a 20–40 % performance improvement with 1 CPU core as I/O core. Further investigation showed the CPU I/O becomes the bottleneck. We tested with multiple CPU cores feeding packets into the GPU and found that the best performance the GPU can achieve for IPv4 is around 175 Mpps, about three times faster than a single core can process, with three CPU cores feeding packets to the GPU.

We also compared IPv4 flow classification, on the GPU accelerated packet processing and multi-core CPU pipelined model, which uses 2 cores for I/O and lookup (as a typical pipeline configuration). On average, two CPU cores achieved ∼21 Mpps throughput for 1 million flows. Compared to the run-to-completion model performance in Fig. 6, the CPU performance with pipe-lined model [18] does not scale well due to the core-to-core communication overhead. *In comparison, one CPU core with GPU acceleration outperforms the CPU pipeline model by 2.5x while freeing one core to perform other useful work.*

VI. Conclusion and Future Works

This paper exploits using integrated GPUs for packet processing workloads to improve overall system performance. We have shown that the proposed GPU accelerated packet processing framework improves the throughput performance by 2–2.5x for many workloads, compared to CPU-only solutions. The improvement is more impressive since the workloads we evaluated only involve lightweight computations. In the future, we plan to explore optimizations in terms of both hardware and software, including flow classification algorithms that fit the GPU processing model to further optimize the performance.

References

[1] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Scaling Up Clustered Network Appliances with ScaleBricks," in *Proc. ACM SIGCOMM*, August 2015.

[2] T.-H. Li, H.-M. Chu, and P.-C. Wang, "IP Address Lookup Using GPU," in *Proc. IEEE HPSR*, July 2013.

[3] Y. Li, D. Zhang, A. X. Liu, and J. Zheng, "GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers," in *Proc. ACM/IEEE ANCS*, October 2013.

[4] J. Zhao, X. Zhang, X. Wang, and X. Xue, "Achieving O(1) IP Lookup on GPU-based Software Routers," in *Proc. ACM SIGCOMM*, August 2010.

[5] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-Accelerated Software Router," in *Proc. ACM SIGCOMM*, August 2010.

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

[7] M. Jamshed *et al.*, "Kargus: A Highly-scalable Software-based Intrusion Detection System," in *Proc. ACM CCS*, October 2012.

[8] A. Kalia, D. Zhou, and M. K. an David G. Anderson, "Raising the Bar for Using GPUs in Software Packet Processing," in *Proc. USENIX NSDI*, May 2015.

[9] OpenCL™ Optimization Guide for Intel® Processor Graphics, `https://software.intel.com/en-us/iocl_opg`.

[10] R. Pagh and F. F. Rodler, "Cuckoo hashing," *ACM Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[11] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Anderson, "Scalable, High Performance Ethernet Forwarding with CucokkSwitch," in *Proc. ACM CoNEXT*, December 2013.

[12] DPDK: Data Plane Development Kit, `http://dpdk.org`.

[13] K. Gupta, J. A. Stuart, and J. D. Owens, "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads," in *Proc. IEEE InPar*, May 2012.

[14] T. Aila and S. Laine, "Understandig the Efficiency of Ray Traversal on GPUs," in *Proc. ACM HPG*, August 2009.

[15] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," in *Proc. IEEE INFOCOM*, March 1998.

[16] IPv4 distribution, `http://psp1.iit.cnr.it/~mcsoft/ast/ast.html`.

[17] IPv6 distribution, `http://dpdk.org/browse/dpdk/plain/app/test/test_lpm6_routes.h?id=v2.0.0`.

[18] Pipelined packet processing model: DPDK packet framework, `http://dpdk.org/doc/guides/prog_guide/packet_framework.html`.