PacketShader: a GPU-Accelerated Software Router

Sangjin Han† Keon Jang† KyoungSoo Park‡ Sue Moon†

†Department of Computer Science, KAIST, Korea
{sangjin, keonjang}@an.kaist.ac.kr, sbmoon@kaist.edu

‡Department of Electrical Engineering, KAIST, Korea
kyoungsoo@ee.kaist.ac.kr

ABSTRACT

We present PacketShader, a high-performance software router framework for general packet processing with Graphics Processing Unit (GPU) acceleration. PacketShader exploits the massively-parallel processing power of GPU to address the CPU bottleneck in current software routers. Combined with our high-performance packet I/O engine, PacketShader outperforms existing software routers by more than a factor of four, forwarding 64B IPv4 packets at 39 Gbps on a single commodity PC. We have implemented IPv4 and IPv6 forwarding, OpenFlow switching, and IPsec tunneling to demonstrate the flexibility and performance advantage of PacketShader. The evaluation results show that GPU brings significantly higher throughput over the CPU-only implementation, confirming the effectiveness of GPU for computation and memory-intensive operations in packet processing.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications; C.2.6 [**Internetworking**]: Routers

General Terms

Design, experimentation, performance

Keywords

Software router, CUDA, GPU

1. INTRODUCTION

PC-based software routers provide a cost-effective packet processing platform with easy extensibility and programmability. Familiar programming environments on general-purpose operating systems allow flexible composition of router applications that meet today's complex traffic engineering demand. Adding to that, modern innovation in commodity hardware continues to drive down the cost per performance, realizing off-the-shelf programmable routers.

While programmability and low cost are the two primary strengths of software routers, keeping them at high speed is still challenging. Existing software routers report near 10 Gbps forwarding performance even on a single machine, but the CPU quickly becomes the bottleneck for more compute-intensive applications. For example, the IPsec performance of RouteBricks degrades by a factor of 4.5 from its 8.7 Gbps[1] IPv4 forwarding throughput with 64B packets [19]. To scale up the computing cycles, one can put more CPUs to a single server or distribute the load to a cluster of machines, but per-dollar performance stays relatively low.

In this work we explore new opportunities in packet processing with Graphics Processing Units (GPUs) to inexpensively shift the computing needs from CPUs for high-throughput packet processing. GPUs offer extreme thread-level parallelism with hundreds of slim cores [20, 42]. Their data-parallel execution model fits nicely with inherent parallelism in most router applications. The memory access latency hiding capability and ample memory bandwidth of GPU can boost many memory-intensive router applications, which heavily rely on lookup in large tables. For compute-intensive applications, the massive array of GPU cores offer an order of magnitude higher raw computation power than CPU. Moreover, the recent trend shows that the GPU computing density improves faster than CPU [42]. Last but not least, GPUs are cheap and readily available.

We present PacketShader, a GPU-accelerated software router framework, that carries the benefit of low cost and high programmability even at multi-10G speed. The main challenge of PacketShader lies in maintaining the high forwarding rate while providing as much processing power for arbitrary router applications. We address the challenge in two parts. First, we implement highly optimized packet I/O engine to eliminate per-packet memory management overhead and to process packets in batch, enabling high-performance packet I/O in user mode. Second, we offload core packet processing operations (such as IP table lookup or IPsec encryption) to GPUs and scale packet processing with massive parallelism. Coupled with I/O path optimization, PacketShader maximizes the utilization of GPUs by exposing as much parallelism as possible in a small time frame. Our design choices allow unprecedented performance advantage. On a single box, PacketShader forwards IPv4 packets at the rate of 40 Gbps for all packet sizes. GPUs bring significant performance improvement for both memory and compute-intensive applications; IPv6 forwarding reaches 38.2 Gbps for 64B packets and the IPsec performance ranges from 10 to 20 Gbps.

PacketShader is the first to demonstrate the potentials of GPUs in the context of multi-10G software routers. We believe GPUs' massively-parallel processing power opens a great opportunity for high-performance software routers with cost effectiveness and full

SIGCOMM'10, August 30–September 3, 2010, New Delhi, India.

[1] We take 24-byte Ethernet overhead into account when we calculate throughput in this paper. We apply the same metric and translate the numbers from other papers.

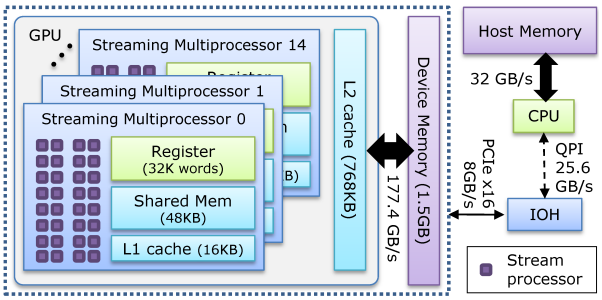


Figure 1: Architecture of NVIDIA GTX480

programmability, and our PacketShader will be a useful stepping stone.

The road map of the paper is as follows. In Section 2 we present an overview of the NVIDIA GPU architecture and explore its potential for packet processing. In Section 3 we describe the hardware and software system setup for our PacketShader. The two key contributions of this work, namely, optimized packet I/O and GPU acceleration, are in Sections 4 and 5, respectively. Section 6 delivers the performance evaluation results. We list current limitations and future directions in Section 7, review related work in Section 8, and conclude in Section 9.

2. GPU AS A PACKET PROCESSOR

GPU has become a powerful computing engine behind scientific computing and data-intensive applications, beyond its original role for graphics rendering. This section explores the potential of GPU for general packet processing.

2.1 GPU Architecture

We begin with a brief introduction on the internals of the NVIDIA GPU that we use in this work. More details are available in [20, 37, 39, 40]. Figure 1 illustrates the architecture of NVIDIA GTX480. It has 15 Streaming Multiprocessors (SMs), each of which consists of 32 Stream Processors (SPs), resulting in 480 cores in total. All threads running on SPs share the same program called *kernel*[2].

An SM works as an independent SIMT (Single Instruction, Multiple Threads) processor. The basic execution unit of SM is a *warp*, a group of 32 threads, sharing the same instruction pointer; all threads in a warp take the same code path. While this lockstep execution is not mandatory, any divergence of code path in a warp should be minimized for optimal performance. For example, when not all 32 threads in a warp agree on the condition of an `if` statement, the threads take both `then` and `else` parts with corresponding masking. Compared to the traditional SIMD (Single Instruction, Multiple Data) architecture where all data elements must be processed in the same way, the SIMT behavior of GPU gives more flexibility to programmers.

The scheduler in an SM holds up to 32 warps and chooses a warp to execute for every instruction issue time; a readily available warp, not having stalls due to register dependency or memory access, is chosen first. Typically, having many GPU threads gives better throughput since memory access latency of a warp can be effectively hidden with execution of other warps [25]. Warp scheduling is done by hardware, incurring zero context-switch overhead.

GTX480 provides high-bandwidth, off-chip GPU memory (device memory of 1.5 GB in Figure 1). For low-latency in-die memory, each SM has 48 KB shared memory (comparable with scratchpad RAM in other architectures), 32K 32-bit registers, and 16KB L1 cache. L2 cache of 768 KB is shared by all SMs.

[2]The term "kernel" should not be confused with operating system kernels.

Buffer size (bytes)	256	1K	4K	16K	64K	256K	1M
Host-to-device	55	185	759	2,069	4,046	5,142	5,577
Device-to-host	63	211	786	1,743	2,848	3,242	3,394

Table 1: Data transfer rate between host and device (MB/s)

The GPU kernel execution takes four steps: (*i*) the DMA controller transfers data from host (CPU) memory to device (GPU) memory, (*ii*) a host program instructs the GPU to launch the kernel, (*iii*) the GPU executes threads in parallel, and (*iv*) the DMA controller transfers resulting data from device memory to host memory.

CUDA (Compute Unified Device Architecture) is NVIDIA's software suite of libraries and a compiler, and it provides an API to GPU functionalities [40]. Programmers write C-like (C with CUDA-specific extensions) kernel code, and then CUDA compiles it and exposes an interface to host programs to launch the kernel.

2.2 GPU Overheads

Typical GPGPU (General-Purpose computing on GPU) applications are data-intensive, handling relatively long-running kernel execution (10-1,000s of ms) and large data units (1–100s of MB) [41]. In the context of software routers, GPU should work with much shorter kernel execution time and smaller data. We check if GTX480 hardware and CUDA library give reasonably small overheads for such fine-grained GPU use.

Kernel launch latency: Every step in a kernel launch contributes to the latency: PCI Express (PCIe) transactions, initial scheduling of threads, synchronization, and notification for completion. We measure the latency to check if it is reasonably small; the GPU launching latency for a single thread is 3.8 μs, and 4.1 μs for 4,096 threads (only 10% increase). We conclude that amortized per-thread kernel launch overhead decreases linearly with the increasing number of threads and eventually becomes negligible.

Data transfer rate: While a PCIe 2.0 x16 link connected to a graphics card offers the theoretical bandwidth of 8 GB/s, the effective bandwidth would be smaller due to PCIe and DMA overheads, especially for small data transfer units. We measure the data transfer rate between host and device memory over different buffer sizes and summarize it in Table 1. The transfer rate is proportional to the buffer size and peaks at 5.6 GB/s for host-to-device and 3.4 GB/s for device-to-host. The table confirms that a PCIe link provides enough bandwidth even for small batch sizes. For example, we can transfer 1 KB of 256 IPv4 addresses (4B each) at 185 MB/s or $185 \div 4 = 48.5$ Mpps for each GPU, which translates to 34.1 Gbps with 64B packets.

2.3 Motivating Example

The processing power of GPU comes from its hundreds of cores. The key insight of this work is that the massive array of GPU cores match the inherent parallelism in stateless packet processing. We process multiple packets at a time and take full advantage of the massive parallelism in GPU.

Figure 2 compares the performance of IPv6 forwarding table lookup (longest prefix matching) with CPU and GPU (with the same algorithm and the forwarding table in Section 6.2.2). The experiment is done with randomly generated IPv6 addresses and does not involve actual packet I/O via Network Interface Cards (NICs). The throughput of GPU is proportional to the level of parallelism; given more than 320 packets an NVIDIA GTX480 outperforms one Intel quad-core Xeon X5550 2.66 GHz CPU and two CPUs with more than 640

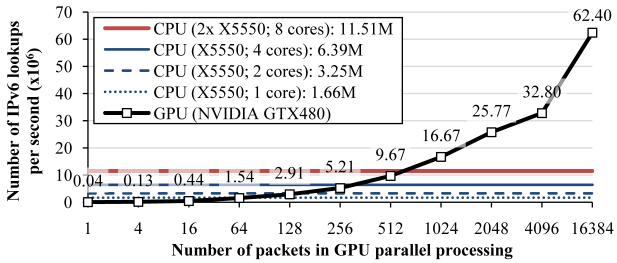


Figure 2: IPv6 lookup throughput of X5550 and GTX480

packets. At the peak performance one GTX480 GPU is comparable to about ten X5550 processors. In contrast, given a small number of packets in a batch GPU shows considerably lower performance compared with CPU. GTX480 processes up to 480 threads at a time and needs more threads to hide memory access latency; having not enough threads, most of GPU resources are underutilized during the execution. The per-batch cost of GPU transaction is another reason for the low performance, as described in Section 2.2.

While the preliminary result of GPU is promising, one natural question arises here: *does collecting hundreds or thousands of packets incur unreasonable latency?* Our answer is "no"; a large number of packets arrive in a fairly small time window on today's high speed links. For example, a thousand 64B packets arrive in only 70 μs on a full-speed 10 GbE link.

2.4 Comparison with CPU

The fundamental difference between CPU and GPU comes from how transistors are composed in the processor. CPUs maximize instruction-level parallelism to accelerate a small number of threads. Most of CPU resources serve large caches and sophisticated control planes for advanced features (e.g., superscalar, out-of-order execution, branch prediction, or speculative loads). In contrast, GPUs maximize thread-level parallelism, devoting most of their die area to a large array of Arithmetic Logic Units (ALUs). GPUs also provide ample memory bandwidth to feed data to a large number of cores. We briefly address the implication of those differences in the context of packet processing.

Memory access latency: For most network applications, memory working set is too big to fit in a CPU cache. While the memory access latency can be potentially hidden with out-of-order execution and overlapped memory references, the latency hiding is limited by CPU resources such as the instruction window size, the number of Miss Status Holding Registers (MSHRs), and memory bandwidth. In our microbenchmark, an X5550 cores can handle about 6 outstanding cache misses in the optimal case, and only 4 misses when all four cores bursts memory references. Unlike CPU, GPU effectively hides memory access latency with hundreds (or thousands) of threads. By having an enough number of threads, memory stalls can be minimized or even eliminated [43].

Memory bandwidth: Network applications tend to exhibit random memory access, such as hash table lookup, quickly exhausting available memory bandwidth. For example, every 4B random memory access consumes 64B of memory bandwidth, which is the size of a cache line in x86 architecture. By offloading memory-intensive operations to GPU, such as IPv6 longest prefix matching, we can benefit from larger memory bandwidth of GPU (177.4 GB/s for GTX480 versus 32 GB/s for X5550). This *additional* memory bandwidth of GPU is particularly helpful when a large portion of CPU's memory bandwidth is consumed by packet I/O.

Item	Specification	Qty	Unit price
CPU	Intel Xeon X5550 (4 cores, 2.66 GHz)	2	$925
RAM	DDR3 ECC 2 GB (1,333 MHz)	6	$64
M/B	Super Micro X8DAH+F	1	$483
GPU	NVIDIA GTX480 (480 cores, 1.4 GHz, 1.5 GB)	2	$500
NIC	Intel X520-DA2 (dual-port 10GbE)	4	$628

Table 2: Test system hardware specification (total $7,000)

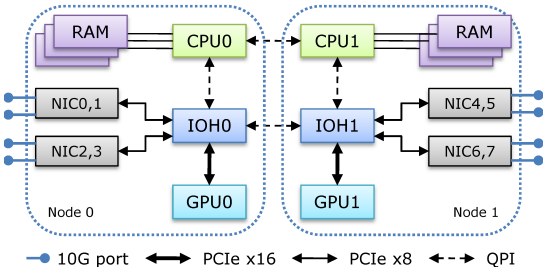


Figure 3: Block diagram of our server

Raw computation capacity: Applications running on software routers are increasingly demanding for compute-intensive operations, such as hashing, encryption, compression, and pattern matching, to name a few. As the bottleneck of software routers lies in CPU, introducing those operations would significantly degrade the performance. GPU can be an attractive source of extra computation power; NVIDIA GTX480 offers an order of magnitude higher peak performance than X5550 in terms of MIPS (Million Instructions Per Second). Moreover, the recent trend shows that the GPU computing density is improving faster than CPU [42].

3. SYSTEM SETUP

This section describes the hardware and software configuration that we use in this paper. We also mention the dual-IOH problem that bounds I/O bandwidth, limiting the performance of our current system.

3.1 Hardware Configuration

We set up our hardware to perform high-speed packet processing with GPUs while reflecting today's commodity hardware trend. Table 2 summarizes the specifications of our server. We use two Intel Nehalem quad-core Xeon X5550 2.66 GHz processors with 1,333 MHz 12 GB DDR3 memory. For GPU acceleration, we use two NVIDIA GTX480 cards, each of which has 480 stream processor cores and 1.5 GB GDDR5 memory. Our choice of GTX480 is the top-of-the-line graphics card at the moment. For network cards, we use four dual-port Intel 82599 X520-DA2 10GbE NICs: eight ports with aggregate capacity of 80 Gbps. The total system (including all other components) costs about $7,000.[3]

Figure 3 shows the components and their interconnection in the server. There are two NUMA (Non-Uniform Memory Access) *nodes* in the server, and each node has a quad-core CPU socket and local memory. The memory controller integrated in a CPU connects three 2 GB DDR3 DRAMs in the triple-channel mode. Each node has an IOH (I/O Hub) that connects peripheral devices to the CPU socket. Each IOH holds three PCIe devices: two dual-port 10GbE NICs on PCIe x8 links and one NVIDIA GTX480 graphics card on a PCIe x16 link.

[3] All prices are from `http://checkout.google.com` on June 2010

Functional bins	Cycles	Our solution
skb initialization	4.9%	Compact metadata (§4.2)
skb (de)allocation	8.0%	Huge packet buffer (§4.2)
Memory subsystem	50.2%	
NIC device driver	13.3%	Batch processing (§4.3)
Others	9.8%	
Compulsory cache misses	13.8%	Software prefetch (§4.3)
Total	100.0%	-

Table 3: CPU cycle breakdown in packet RX

3.2 Dual-IOH Problem

Our system uses four NICs and two GPUs which require 64 PCIe lanes in total. Since an Intel 5520 IOH chipset provides only 36 PCIe lanes, we adopt a motherboard with two IOH chipsets.

However, we have found that there is asymmetry of PCIe data transfer performance with the dual-IOH motherboard. We see much lower empirical bandwidth of device-to-host transfer than that of host-to-device transfer. Table 1 in the previous section shows that data copy from a GPU to host memory is slower than the opposite direction. Similarly for NICs, we see higher TX throughput than RX as shown in Figure 6 in Section 4.6.

The throughput asymmetry is specific to motherboards with dual 5520 chipsets. Motherboards from other vendors with two IOHs have the same problem [7], and we confirm that motherboards with a single IOH do not show any throughput asymmetry [23]. We are investigating the exact cause of the problem and planning to check if motherboards based on AMD chipsets have the same problem.

The dual-IOH problem limits the packet I/O performance in our system, and the limited I/O throughput bounds the maximum performance of our applications in Section 6.

3.3 Software Configuration

We have installed 64-bit Ubuntu Linux 9.04 server distribution with unmodified Linux kernel 2.6.28.10 in the server. Our packet I/O engine (Section 4) is based on ixgbe 2.0.38.2 device driver for Intel 10 GbE PCIe adapters. For GPU, we use the device driver 195.36.15 and CUDA SDK 3.0.

4. OPTIMIZING PACKET I/O ENGINE

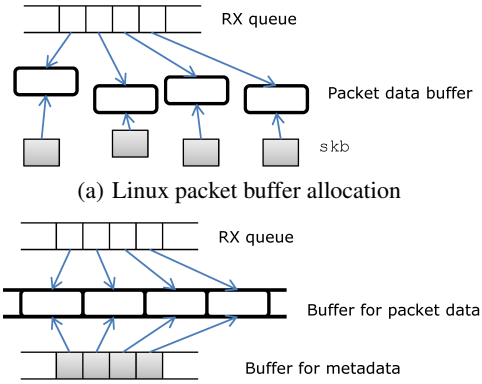
High-speed software routers typically spend a large fraction of CPU cycles on packet reception and transmission via NICs, the common part of all router applications. For example, RouteBricks reports that 66% of total cycles are spent on packet I/O for IPv4 forwarding [19]. This means that even if we eliminate other packet handling costs with the help of GPU, the expected improvement would not exceed 50% according to Amdahl's law.

In order to achieve multi-10G packet I/O performance in the software router, we exploit pipelining and batching aggressively. In this work we focus on the basic interfacing with NICs and leave other advanced features, such as intermediate queueing and packet scheduling, for future work.

4.1 Linux Network Stack Inefficiency

Network stacks in the OS kernel maintain packet buffers. A packet buffer is a basic message unit passed across network layers. For example, Linux allocates two buffers, an skb holding metadata and a buffer for actual packet data, for each packet. This per-packet buffer allocation applies to Click [30] as well since it relies on Linux data structures. We observe two problems arising here:

- Frequent buffer allocation and deallocation stress the memory subsystem in the kernel. In multi-10G networks, this implies tens of millions of buffer allocations per second.
- The metadata size in skb is too large (208 bytes long in Linux 2.6.28), as it holds information required by all protocols in various layers. It is overkill for 64B packets.



(a) Linux packet buffer allocation

(b) Huge packet buffer allocation

Figure 4: Comparison of packet buffer allocation schemes

To quantify where most CPU cycles are spent, we measure the CPU consumption of the packet reception process. We have the unmodified ixgbe NIC driver receive 64B packets and silently drop them. Table 3 shows the breakdown of the CPU cycles in the packet RX process. We see that skb-related operations take up 63.1% of the total CPU usage: 4.9% on initialization, 8.0% on allocation and deallocation wrapper functions, and 50.2% on base memory subsystem (the slab allocator [16] and the underlying page allocator) to handle memory allocation and deallocation requests.

Cache invalidation with DMA causes compulsory cache misses accounting for 13.8% in the table. Whenever a NIC receives or transmits packets, it accesses the packet descriptors and data buffers with DMA. Because DMA transactions invalidate corresponding CPU cache lines for memory consistency, the first access to memory regions mapped for DMA causes compulsory cache misses.

4.2 Huge Packet Buffer

As described above, per-packet buffer allocation (in Figure 4(a)) causes significant CPU overhead. To avoid the problem, we implement a new buffer allocation scheme called *huge packet buffer* (Figure 4(b)). In this scheme, the device driver does not allocate skb and a packet data buffer for each packet. Instead, it allocates two huge buffers, one for metadata and the other for packet data. The buffers consist of fixed-size cells, and each cell corresponds to one packet in the RX queue. The cells are reused whenever the circular RX queues wrap up. This scheme effectively eliminates the cost of per-packet buffer allocation. Huge packet buffers also reduce per-packet DMA mapping cost (listed in Table 3 as part of NIC device driver cost), which translates the host memory address into I/O device memory address so that NICs can access the packet data buffer. Instead of mapping a small buffer for every packet arrival, we have the device driver map the huge packet buffer itself for efficient DMA operations.

To reduce the initialization cost of the metadata structure, we keep the metadata as compact as possible. Most fields in skb are unnecessary, for packets in software routers do not traverse the Linux network stack. We have removed the unused fields and the resulting metadata cell is only 8 bytes long rather than 208 bytes. Each cell of the packet data buffer is 2,048-byte long, which fits for the 1,518-

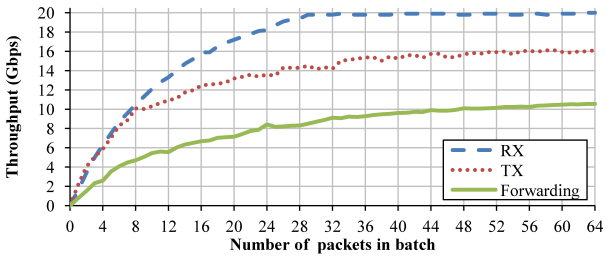


Figure 5: Effect of batch processing with a single core and two 10 GbE ports. All packets are 64B.

byte maximum Ethernet frame size and works around the NIC requirement of 1,024-byte data alignment.

4.3 Batch Processing

Batch processing of multiple packets reduces per-packet processing overhead significantly. Batching can be applied at every step in packet processing: (*i*) in hardware: NICs aggregate multiple packets into a single PCIe transaction for better I/O bandwidth utilization, (*ii*) in device driver: the amortized cost of per-packet bookkeeping operations decreases, and (*iii*) in applications: processing multiple packets achieves smaller instruction footprint with reduced function call overheads and synchronization operations. We further improve the batching approach of RouteBricks [19] as follows.

- We extend batch processing to the application (packet processing) level. In contrast, Click [30] handles multiple packets at a time only at NIC and device driver levels, but processes packets one-by-one at the application level.
- To maximize benefit from huge packet buffers, our packet I/O engine performs aggressive software prefetch. The device driver prefetches the packet descriptor and packet data of the next packet into CPU cache in parallel while processing the packet. This prefetch of consecutive packets eliminates the compulsory cache miss latency.

When we pass batched RX packets to user-level applications, we copy the data in the huge packet buffer into a consecutive user-level buffer along with an array of offset and length for each packet. The rationale for copying instead of zero-copy techniques (e.g., shared huge packet buffer between kernel and user) is for better abstraction. Copying simplifies the recycling of huge packet buffer cells and allows flexible usage of the user buffer, such as manipulation and split transmission of batched packets to multiple NIC ports. We find that the copy operation makes little impact on performance; it does not consume additional memory bandwidth since the user buffer is likely to reside in CPU cache and takes less than 20% of CPU cycles out of total packet I/O processing.

Figure 5 shows the RX, TX, and minimal forwarding (RX + TX) performance with 64B packets using a single CPU core. We see that the throughput improves as the number of packets in a batch increases, but the performance gain stalls after 32 packets. While the packet-by-packet approach (with the batch size of 1) handles only 0.78 Gbps, batch processing achieves 10.5 Gbps packet forwarding performance with the batch size of 64 packets, resulting in the speedup of 13.5.

4.4 Multi-core Scalability

Packet I/O on multi-core systems raises two performance issues: (i) load balancing between CPU cores and (ii) linear performance scalability with additional cores. To address these challenges, recent NICs support core-aware RX and TX queues with Receive-Side Scaling (RSS) [12]. RSS evenly distributes packets across multiple RX queues by hashing the five-tuples (source and destination addresses, ports, and a protocol) of a packet header. Each RX and TX queue in a NIC maps to a single CPU core, and the corresponding CPU core accesses the queue exclusively, eliminating cache bouncing and lock contention caused by shared data structures. Our packet I/O engine takes advantage of RSS and multiple queues in a NIC.

However, we find that core-aware multi-queue support alone is not enough to guarantee the linear performance scalability with the number of CPU cores. In our microbenchmark with eight cores, per-packet CPU cycles increase by 20% compared to the single-core case.

After careful profiling, we have identified two problems. First, some per-queue data that are supposed to be private bounce between multiple CPU caches. It happens when the data of different queues accidently share the same cache line (64B per line in the x86 architecture). We eliminate this behavior, known as *false sharing* [50], by aligning every starting address of per-queue data to the cache line boundary. The second problem comes from accounting. Whenever the device driver receives or transmits a packet, it updates per-NIC statistics. These globally shared counters stress CPU caches as every statistics update is likely to cause a coherent cache miss. We solve the problem by maintaining per-queue counters rather than per-NIC ones so that multiple CPU cores do not contend for the same data. Upon a request for per-NIC statistics (from commands `ifconfig` or `ethtool`), the device driver accumulates all per-queue counters. This on-demand calculation for a rare reference to NIC statistics keeps frequent statistics updates cheap.

4.5 NUMA Scalability

In a NUMA system, memory access time depends on the location of physical memory (see Figure 3). Locally-connected memory allows direct access while remote memory access requires an extra hop via the hosting CPU node. Also, DMA transactions from a device to a remote node traverse multiple IOHs and reduce I/O performance. In our testing, we see that node-crossing memory access shows 40-50% increased access time and 20-30% lower bandwidth compared to in-node access.

To maximize packet I/O performance in the NUMA system, we make the following two choices. First, we carefully place all data structures in the same node where they are used. The packet descriptor arrays, huge packet buffers, metadata buffers, and statistics data of NICs are allocated in the same node as the receive NICs. Second, we remove node-crossing I/O transactions caused by RSS. By default RSS distributes packets to all CPU cores and some cross the node boundary. For example, half of the packets received by NICs in node 0 in Figure 3 would travel to the memory in node 1. To eliminate these crossings, we configure RSS to distribute packets only to those CPU cores in the same node as the NICs. We set the number of RX queues as the number of corresponding CPU cores and map the RX interrupts for the queues to those CPU cores in the same node. With these modifications, interrupts, DMA transactions, and PCIe register I/O for packet RX do not cross the node boundary.

With NUMA-aware data placement and I/O, we see about 60% performance improvement over NUMA-blind packet I/O. NUMA-blind packet I/O limits the forwarding performance below 25 Gbps while NUMA-aware packet I/O achieves around 40 Gbps. Our result contradicts the previous report: RouteBricks concludes that NUMA-aware data placement is not essential in their work [19] with multi-gigabit traffic. We suspect that additional memory access latency

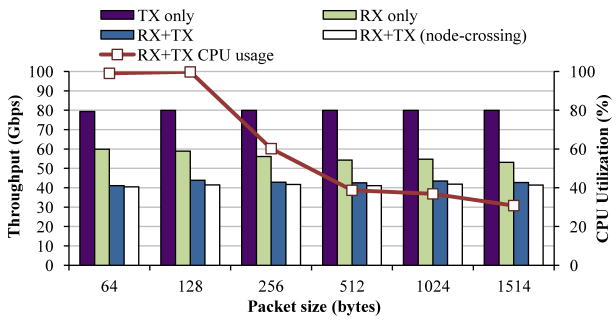


Figure 6: Performance of our packet I/O engine

caused by NUMA-blind data placement is almost hidden with multi-gigabit traffic, but not with multi-10G traffic.

4.6 Packet I/O Performance

For the evaluation of our packet I/O engine, we implement a simple user-level program that repeatedly receives, transmits, and forwards packets without IP table lookup. Figure 6 summarizes the performance of our packet I/O engine with all the techniques introduced in this section. The results shown here are encouraging. The TX performance comes close to the line rate, with 79.3 Gbps for 64B packets and 80.0 Gbps for 128B or larger packets. The RX performance ranges from 53.1 Gbps to 59.9 Gbps depending on the packet size. This performance asymmetry seems to stem from the DMA inefficiency from a device to host memory, as described in Section 3.2.

With RX and TX together, the minimal forwarding performance (without IP table lookup) stays above 40 Gbps for all packet sizes. By comparison, RouteBricks forwards 64B packets at 13.3 Gbps or 18.96 Mpps running in kernel mode on a slightly faster machine with dual quad-core 2.83 GHz CPUs. Our server outperforms RouteBricks by a factor of 3 achieving 41.1 Gbps or 58.4 Mpps for the same packet size even though our experiment is done in user mode.

Bars marked as "node-crossing" represent the case that all packets received in one NUMA node are transmitted to ports in the other node. Even for this worst case the throughput still stays above 40 Gbps, implying our packet I/O engine is scalable with NUMA architecture.

We identify the bottleneck limiting the throughput around 40 Gbps, which could reside in either CPU, memory bandwidth, or I/O. Though we see the full CPU usage for 64B and 128B packets, CPU is not the bottleneck. We run the same test with only four CPU cores (two for each node) and get the same forwarding performance. As implied in Figure 5, the CPU usage is elastic with the number of packets for each fetch; the average batch size was 13.6 with 8 cores and 63.0 with 4 cores. To see whether the memory bandwidth is the bottleneck, we run the same experiments by having background processes consume additional memory bandwidth, but we see the same performance.

We conclude that the bottleneck lies in I/O. Considering the throughput asymmetry of RX and TX and the fact that individual links (QPI and PCIe) provide enough bandwidth, we suspect that the dual-IOH problem described in Section 3.2 bounds the maximum packet I/O performance around 40 Gbps in our system.

5. GPU ACCELERATION FRAMEWORK

Our GPU acceleration framework provide a convenient environment to write packet processing *applications*, maximizing synergy with our highly optimized packet I/O engine. This section describes the brief design of the framework.

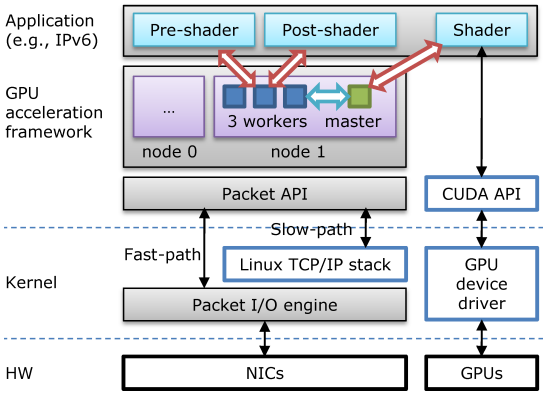


Figure 7: PacketShader software architecture

5.1 Overall Structure

PacketShader is a software router framework that combines the GPU acceleration template and our highly optimized packet I/O engine. Figure 7 depicts the simplified architecture of PacketShader (gray blocks indicate our implementation). PacketShader is a multi-threaded program running in user mode. For packet I/O, it invokes the packet API consisting of wrapper functions to kernel-level packet I/O engine. A packet processing application runs on top of the framework and is mainly driven by three callback functions (a pre-shader, a shader, and a post-shader).

Currently, the performance of CUDA programs degrades severely when multiple CPU threads access the same GPU, due to frequent context switching overheads [29]. In order to avoid the pathological case, PacketShader divides the CPU threads into *worker* and *master* threads. A master thread communicates exclusively with a GPU in the same node for acceleration, while a worker thread is responsible for packet I/O and requests the master to act as a proxy for communication with the GPU. Each thread has a one-to-one mapping to a CPU core and is hard-affinitized to the core to avoid context switching and process migration costs [21].

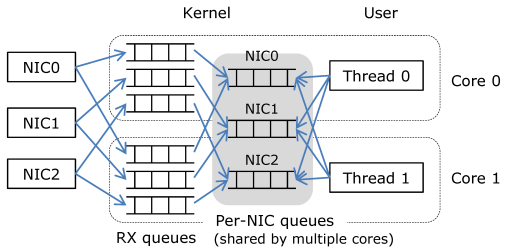
PacketShader partitions the system into NUMA nodes so that each of them could process packets independently. In each node a quad-core CPU runs three worker threads and one master thread. Those workers communicate only with the local master to avoid expensive node-crossing communication. Once a worker thread receives packets, all processing is done by the CPU and the GPU within the same node. The only exception is to forward packets to ports in the other node, but this process is done by DMA, not CPU.

5.2 User-level Interface to Packet I/O Engine

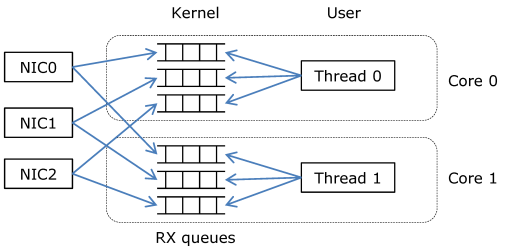
PacketShader runs in user mode, not in kernel, to take advantage of user-level programming: friendly development environments, reliability with fault isolation, and seamless integration with third-party libraries (e.g., CUDA or OpenSSL). User-level packet processing, however, imposes a technical challenge: performance. For example, a user-mode Click router is reported to be three times slower than when it runs in kernel space [31]. We list three major issues and our solutions for high-performance packet processing in user mode.

Minimizing per-packet overhead: The simplest packet I/O scheme, one system call for each packet RX and TX, introduces significant user-kernel mode switching overheads. PacketShader batches multiple packets over a single system call to amortize the cost of per-packet system call overhead.

Better coupling of queues and cores: Existing user-level packet I/O libraries, such as libpcap [26], exposes a per-NIC interface to user



(a) Existing per-NIC queue scheme



(b) Our multiqueue-aware packet I/O scheme

Figure 8: User-level packet I/O interfaces

applications. This scheme significantly degrades the performance on systems equipped with multi-queue NICs and multi-core CPUs, as shown in Figure 8(a). Kernel and user threads on multiple CPU cores have to contend for the shared per-NIC queues. These multiplexing and de-multiplexing of received packets over the shared queues cause expensive cache bouncing and lock contention.

PacketShader avoids this problem by using an explicit interface to individual queues (Figure 8(b)). A virtual interface, identified with a tuple of (NIC id, RX queue id), is dedicated to a user-level thread so that the thread directly accesses the assigned queue. The virtual interfaces are not shared by multiple cores, eliminating the need of sharing and synchronization of a single queue. The user thread fetches packets from multiple queues in a round-robin manner for fairness.

Avoiding receive livelock: Click and other high-performance packet processing implementations avoid the receive livelock problem [34] by polling NICs to check packet reception rather than using interrupts [17, 18, 30]. However, we note two problems with this busy-wait polling: (i) Extended period of full CPU usage prevents the chance to save electricity in an idle state, which can be up to a few hundreds of watts for a modern commodity server. (ii) Polling in kernel can starve user processes (e.g., a BGP daemon process) running on the same CPU core.

Linux NAPI [44], a hybrid of interrupt and polling in Linux TCP/IP stack, effectively prevents TCP/IP stack in kernel from starvation. PacketShader can not directly benefit from NAPI, however, since NAPI protects only kernel context[4]. User context, which has the lowest scheduling priority in the system, may always be preempted by the kernel (hardware RX interrupt or *softirq*, the bottom-half handling of received packets).

To avoid receive livelock problem in user context, PacketShader actively takes control over switching between interrupt and polling. In the interrupt-disabled state, PacketShader repeatedly fetches packets. When it drains all the packets in the RX queue, the thread blocks and enables the RX interrupt of the queue. Upon receiving packets,

[4]NAPI also indirectly saves user programs using TCP from starvation because the packet arrival rate is suppressed by congestion control followed by packet drop.

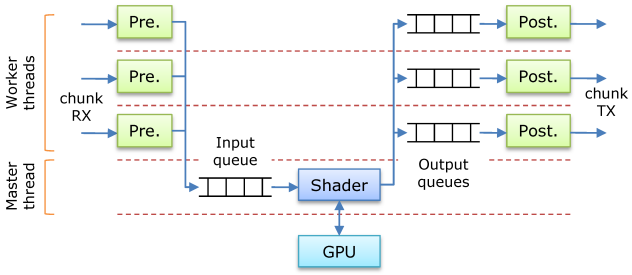


Figure 9: Basic workflow in PacketShader

the interrupt handler wakes up the thread, and then the interrupt is disabled again. This scheme effectively eliminates receive livelock as packet RX only happens with the progress of PacketShader.

5.3 Workflow

We define *chunk* as a group of packets fetched in a batch of packet reception. The chunk size is not fixed but only capped; we do not intentionally wait for the fixed number of packets. Chunk also works as the minimum processing unit for GPU parallel processing in our current implementation. By processing multiple packets pending in RX queues, PacketShader adaptively balances between small parallelism for low latency and large parallelism for high throughput, according to the level of offered load.

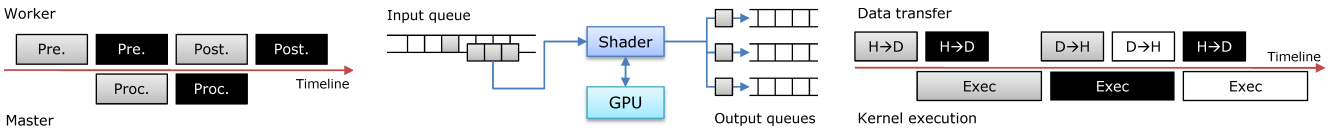
PacketShader divides packet processing into three steps: *pre-shading*, *shading*[5], and *post-shading*. Pre-shading and post-shading run on worker threads and perform the actual packet I/O and other miscellaneous tasks. Shading occurs on master threads and does GPU-related tasks. Each step works as follows:

- **Pre-shading**: Each worker thread fetches a chunk of packets from its own RX queues. It drops any malformed packets and classifies normal packets that need to be processed with GPU. It then builds data structures to feed input data to the GPU. For example, for IPv4 forwarding, it collects destination IP addresses from packet headers and makes an array of the collected addresses. Then it passes the input data to the *input queue* of a master thread.
- **Shading**: The master thread transfers the input data from host memory to GPU memory, launches the GPU kernel, and takes back the results from GPU memory to host memory. Then it places the results back to the *output queue* of the worker thread for post-shading.
- **Post-shading**: A worker thread picks up the results in its output queue, and modifies, drops, or duplicates the packets in the chunk depending on the processing results. Finally, it splits the packets in the chunk into destination ports for packet transmission.

Figure 9 shows how the worker and master threads collaborate on GPU acceleration in PacketShader. Communication between threads is done via the input queue of a master thread and output queues of worker threads. Having per-worker output queues relaxes cache bouncing and lock contention by avoiding 1-to-N sharing. However, we do not apply the same technique to the input queue in order to guarantee fairness between worker threads.

Another issue with communication between CPU cores is cache

[5]In computer graphics, a shader is a small program running on GPUs for rendering. A shader applies transformation to a large set of vertices or pixels in parallel. We name PacketShader after this term since it enables GPUs to process packets in parallel.



(a) Chunk pipelining (b) Gather/scatter (c) Concurrent copy and execution

Figure 10: Optimizations on GPU acceleration

migration of the input and output data. The pre-shading step builds the input data that the CPU core loads onto the private cache. If a master thread accesses the input data, corresponding cache lines would migrate to the master thread's CPU cache. Since cache migration is expensive on multi-core systems, we prevent the master thread from accessing the input data itself. When the master thread is notified of input data by a worker thread, it immediately transfers it to GPU by initiating DMA without touching the data itself. We apply the same policy to the output data as well.

PacketShader preserves the order of packets in a flow with RSS. RSS distributes received packets into worker threads on a flow-basis; packets in the same flow are processed by the same worker thread. PacketShader retains the initial order of packets in a chunk, and all queues enforce First-In-First-Out (FIFO) ordering.

5.4 Optimization Strategies

PacketShader exploits optimization opportunities in the basic workflow. First, we pipeline the processing steps to fully utilize each worker thread. Second, we have the master thread process multiple chunks fed from worker threads at a time to expose more parallelism to GPU. Third, we run the PCIe data transfer in parallel to the GPU kernel execution.

Chunk Pipelining: After a worker thread performs the pre-shading step and passes the input data to its master, it waits until the master process finishes the shading step. This underutilization of worker threads degrades the system performance when the volume of input traffic is high.

To remedy this behavior, PacketShader pipelines chunk processing in the worker threads as shown in Figure 10(a). Once a worker thread passes input data to its master, it immediately performs the pre-shading step for another chunk until the master returns the output data of the first chunk.

Gather/Scatter: As PacketShader has multiple worker threads per master, the input queue can fill with chunks if the GPU is overloaded. Figure 10(b) shows how PacketShader processes multiple chunks in the queue in the shading step; a master thread dequeues multiple input data from its input queue and pipelines copies of input data (gather), processes them with the GPU at a time, and splits the results into the output queues of workers from which the chunks came (scatter).

This optimization technique is based on the observations from Section 2: (*i*) having more GPU threads per GPU kernel launch amortizes per-packet kernel launch overhead, (*ii*) the use of many GPU threads can effectively hide memory access latency without extra scheduling overhead. Additionally, pipelined copies yield better PCIe data transfer rate. Our gather/scatter mechanism gives the GPU more parallelism and improves the overall performance of the system.

Concurrent Copy and Execution: In the shading step, the data transfers and the GPU kernel execution of a chunk have dependency on each other and are serialized. In the meantime, NVIDIA GPUs support copying data between host and device memory while executing a GPU kernel function. This optimization is called "concurrent copy and execution" and is popular for many GPGPU applications [40]. This technique can improve the throughput of the GPU by a factor of two, if data transfer and GPU kernel execution completely overlap.

CUDA supports a *stream* to allow a CPU thread have multiple device contexts. With multiple streams PacketShader can overlap data transfers and kernel launches for consecutive chunks, as shown in Figure 10(c). However, we find that using multiple streams significantly degrades the performance of lightweight kernels, such as IPv4 table lookup. Since having multiple streams adds non-trivial overhead for each CUDA library function call, we selectively use this technique for the IPsec application in Section 6.

5.5 GPU Programming Considerations

All GPU kernels used in Section 6 are the straightforward porting of CPU code (one exception is IPsec; we have made an effort to maximize the usage of in-die memory for optimal performance). In general, turning a typical C program into a correct GPU kernel requires only little modification. However, efficient implementation of GPU programs requires the understanding of characteristics and trade-offs of GPU architecture. We briefly address the general considerations of GPU programming in the context of packet processing acceleration.

What to offload: The offloaded portion to GPU should have non-trivial costs to compensate the overheads of copy from/to GPU memory and kernel launch. Computation and memory-intensive algorithms with high regularity suit well for GPU acceleration.

How to parallelize: Most applications of software routers operate on packet headers. In this case, the most intuitive way to parallelize packet processing is to map each packet into an independent GPU thread; CPU code for per-packet processing can be easily ported to GPU. If there exists exploitable parallelism within a packet (e.g., parallel pattern matching or block cipher operation), we can map each packet into multiple threads for optimal performance with fine-grained parallelism (see our IPsec implementation in Section 6.2.4).

Data structure usage: Simple data structures such as arrays and hash tables are recommended in GPU. Data structures highly scattered in memory (e.g., balanced trees) would make update the data difficult and degrade the performance due to small caches in GPU and uncoalesced memory access pattern [38].

Divergency in GPU code: For optimal performance, the SIMT architecture of CUDA demands to have minimal code-path divergence caused by data-dependent conditional branches within a warp (a group of 32 threads; see Section 2.1). We believe that inherently divergent operations in packet processing are rare or at least avoidable with choices of appropriate algorithms and data structures; all GPU kernels used in this work have no or little code-path divergence. To avoid warp divergence for differentiated packet processing (e.g., packet encryption with different cipher suites), one may classify and sort packets to be grouped into separate warps in GPU so that all threads within a warp follows the same code path.

We expect that these requirements of GPU programming will be

much relaxed considering the current evolution trends of GPU architectures. For example, Intel has recently introduced Larrabee, the many-core x86 GPU architecture with full cache coherency [45]. Its MIMD architecture (multiple-instruction, multiple-data) will ease the current divergency issue of PacketShader. AMD Fusion will integrate CPU and GPU into a single package [1], with reduced GPU communication overheads and a unified memory space shared by CPU and GPU.

6. EVALUATION

In order to demonstrate the performance and flexibility of PacketShader, we implement four applications on top of PacketShader: IPv4 and IPv6 forwarding, OpenFlow switch, and IPsec tunneling. We focus on the data-path performance in our evaluation and assume IP lookup tables, flow tables, and cipher keys are static.

6.1 Test Methodology

We have implemented a packet generator that can produce up to 80 Gbps traffic with 64B packets. It is based on our optimized packet I/O engine, and generates packets with random destination IP addresses and UDP port numbers (so that IP forwarding and OpenFlow look up a different entry for every packet). The generator is directly connected to the PacketShader server via eight 10 GbE links, and it works as both a packet source and a sink.

We implement each application in two modes, the CPU-only mode and the CPU+GPU mode, to evaluate the effectiveness of GPU acceleration. The experiments are performed on the server with eight CPU cores and two GPUs, as described in Section 3.1. The CPU-only mode runs eight worker threads rather than six workers and two masters in the CPU+GPU mode since there is no shading step in the CPU-only mode.

6.2 Applications

6.2.1 IPv4 Forwarding

The performance of forwarding table lookup is typically limited by the number of memory access because the table is too big to fit in the CPU cache. The required number of memory access depends on the lookup algorithm and the table size. In our implementation we use *DIR-24-8-BASIC* in [22]. It requires one memory access per packet for most cases, by storing next-hop entries for every possible 24-bit prefix. If a matching prefix is longer than 24 bits, it requires one more memory access. To measure the performance under a realistic condition, we populate the forwarding table with the BGP table snapshot collected on September 1, 2009 from RouteViews [14]. The number of unique prefixes in the snapshot is 282,797, and only 3% percent of the prefixes are longer than 24 bits.

GPU-accelerated IPv4 table lookup runs in the following order. In the pre-shading step, a worker thread fetches a chunk of packets. It collects packets that require slow-path processing (e.g., destined to local, malformed, TTL expired, or marked as wrong IP checksum by NICs) and passes them onto Linux TCP/IP stack. For the remaining packets it updates TTL and checksum fields, gathers destination IP addresses into a new buffer, and passes the pointer to the master thread. In the shading step, the master thread transfers the IP addresses into the GPU memory and launches the GPU kernel to perform the table lookup. The GPU kernel returns a pointer of the buffer holding the next-hop information for each packet. The master thread copies the result from device memory to host memory, and then passes it to the worker thread. In the post-shading step, the worker thread distributes packets into NIC ports based on the forwarding decision.

6.2.2 IPv6 Forwarding

IPv6 forwarding requires more memory access than IPv4 forwarding due to the 128-bit width of IPv6 addresses. For IPv6 table lookup we adopt the algorithm in [55], which performs binary search on the prefix length to find the longest matching prefix. It requires seven memory accesses, thus memory bandwidth and access latency limit the IPv6 forwarding performance.

IPv6 is not popular in practice yet and the number of routing prefixes is much smaller than that of IPv4. Although forwarding table lookup requires a constant number of memory access, a small lookup table would give the CPU-only approach unfair advantage because the small memory footprint would fit in the CPU cache. Instead, we randomly generate 200,000 prefixes for our experiments. IPv6 forwarding works similarly to IPv4, except that a wide IPv6 address causes four times more data to be copied into the GPU memory.

6.2.3 OpenFlow Switch

OpenFlow is a framework that runs experimental protocols over existing networks [13,33]. Packets are processed on a flow basis and do not interfere with other packets of existing protocols. OpenFlow consists of two components, the OpenFlow controller and the OpenFlow switch, running on separate machines in general. The OpenFlow controller, connected via secure channels to switches, updates the flow tables and takes the responsibility of handling unmatched packets from the switches. The OpenFlow switch is responsible for packet forwarding driven by flow tables.

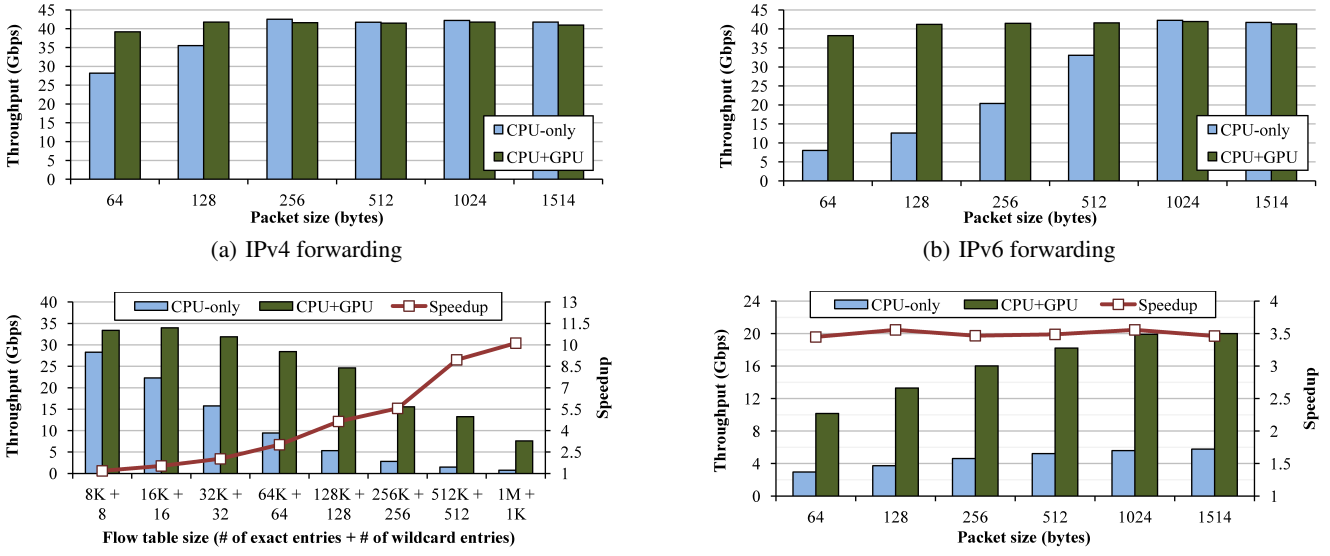
We focus on the OpenFlow switch, based on the OpenFlow 0.8.9r2 specification [10]. The OpenFlow switch maintains the exact-match and the wildcard-match tables. Exact-match entries specify all ten fields in a tuple, which is used as the flow key. In contrast, wildcard match entries specify only some fields (bitmask is also available for IP addresses). An exact-match entry always takes precedence over a wildcard entry. All wildcard entries are assigned a priority.

When a packet arrives, our OpenFlow switch extracts the ten-field flow key from the packet header. For an exact table lookup, the switch matches the flow key against the exact-match entries in the hash table. For a wildcard-table lookup, our switch performs linear search through the table, as the reference implementation does [9], while hardware implementation typically uses TCAM. Our CPU-only OpenFlow switch implements all operations (flow key extraction, hash value calculation and lookup for exact entries, linear search for wildcard matching, and follow-up actions) in CPU. In the GPU-accelerated implementation, we offload hash value calculation and the wildcard matching to GPU, while leaving others in CPU for load distribution.

6.2.4 IPsec Gateway

IPsec is widely used to secure VPN tunnels or for secure communication between two end hosts. Since cryptographic operations used in IPsec are highly compute-intensive, IPsec routers often use hardware accelerator modules. The computational requirement of IPsec makes GPU attractive as it is well-suited for cryptographic operation [24].

For IPsec evaluation, we choose AES-128-CTR for block cipher and SHA1 for Hash-based Message Authentication Code (HMAC). For the CPU+GPU mode, we offload AES and SHA1 to GPU, while leaving other IPsec operations in CPU. The CPU-only approach uses highly optimized AES and SHA1 implementations using SSE instructions for fair comparison. Our implementation runs Encapsulation Security Payload (ESP) IPsec tunneling mode. While this mode increases the packet size with the extra IP header, the ESP header,



(a) IPv4 forwarding

(b) IPv6 forwarding

(c) OpenFlow switch (with 64B packets)

(d) IPsec gateway

Figure 11: Performance measurement of PacketShader on CPU-only and CPU+GPU modes

and the padding, we take input throughput as a metric rather than output throughput.

Our GPU implementation exploits two different levels of parallelism. For AES we maximize parallelism for high performance at the finest level; we chop packets into AES blocks (16B) and map each block to one GPU thread. However, SHA1 cannot be parallelized at the SHA1 block level (64B) due to data dependency between blocks; we parallelize SHA1 at the packet level.

6.3 Throughput

Figure 11 depicts the performance of PacketShader for four applications. We measure the throughput over different packet sizes, except for the OpenFlow switch.

IPv4 and IPv6 Forwarding: Figures 11(a) and 11(b) show IP packet forwarding performance. For all packet sizes, the CPU+GPU mode reaches close to the maximum throughput of 40 Gbps, bounded by packet I/O performance in Figure 6. PacketShader runs at 39 Gbps for IPv4 and 38 Gbps for IPv6 with 64B packets, which are slightly lower than 41 Gbps of minimal forwarding performance. This is because IOH gets more overloaded due to copying IP addresses and lookup results between host memory and GPU memory. We expect to see better throughput, once the hardware problem (in Section 3.2) is fixed.

We find that GPU-acceleration significantly boosts the performance of memory-intensive workloads. The improvement is especially noticeable with IPv6 since it requires more memory access (seven for each packet) than IPv4 (one or two). A large number of threads running on GPU effectively hide memory access latency, and the large GPU memory bandwidth works around the limited host memory bandwidth.

OpenFlow Switch: For OpenFlow switch, we measure the performance variation with different table sizes (Figure 11(c)). CPU+GPU mode outperforms CPU-only mode for all configurations. The performance improvement comes from offloading the hash value computation for small table sizes. Wildcard-match offload becomes dominant as the table size grows.

We compare our PacketShader with the OpenFlow implementation on a NetFPGA card [36]. The NetFPGA implementation is capable of 32K + 32 table entries at maximum, showing 4 Gbps line-rate performance. For the same configuration, PacketShader runs at 32 Gbps, which is comparable with the throughput of eight NetFPGA cards.

IPsec Gateway: Figure 11(d) shows the results of the IPsec experiment. The GPU acceleration improves the performance of the CPU-only mode by a factor of 3.5, regardless of packet sizes. The CPU+GPU throughput for 64B packets is around 10.2 Gbps, and 20.0 Gbps for 1514B packets. PacketShader outperforms RouteBricks by a factor of 5 for 64B packets. RouteBricks achieves IPsec performance 1.9 Gbps for 64B traffic and 6.1 Gbps for larger packets [19]. We note that the IPsec performance of PacketShader is comparable or even better than that of commercial hardware-based IPsec VPN appliances [3].

We suspect that our current performance bottleneck lies in the dual-IOH problem (Section 3.2) again for the CPU+GPU case, as CPUs have not been 100% utilized. In IPsec encryption, entire packet payloads and other metadata (such as keys and IVs) are transmitted from/to GPU, weighing on the burden of IOHs. Experiments done without packet I/O, thus with less traffic through IOHs, show that the performance of two GPUs scales up to 33 Gbps, which implies GPU is not a bottleneck as well.

OpenFlow and IPsec represent compute-intensive workloads of software routers in our work. We have confirmed that compute-intensive applications can benefit from GPU as well as memory-intensive applications.

6.4 Latency

Some techniques used in our work, namely batched packet I/O in Section 4.3 and parallel packet processing with GPU in Section 5, may affect the latency of PacketShader. To quantify it, we measure the average round-trip latency for IPv6 forwarding. The measurement is done at the generator with timestamped 64B packets, over a range of input traffic levels.

Figure 12 shows the measured roundtrip latency of three cases: (*i*) the CPU-only mode without batch processing, (*ii*) the CPU-only mode with batch, and (*iii*) the CPU+GPU mode with both batch and parallelization. Comparing the former two cases, batched packet I/O

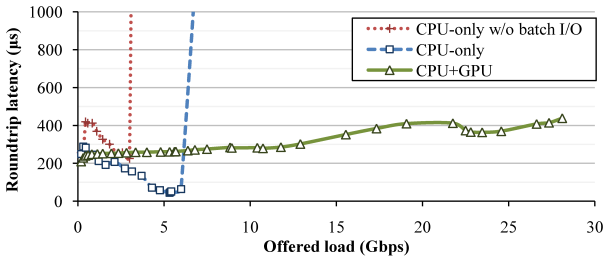


Figure 12: Average roundtrip latency for IPv6 forwarding

shows even lower latency; better forwarding rate with batched I/O effectively reduces the queueing delay of packets. We suspect that higher latency numbers at low input traffic rate are caused by the effect of interrupt moderation in NICs [28]. Comparing the latter two cases, GPU acceleration causes higher latency due to GPU transaction overheads and additional queueing (input and output queues in Figure 9), yet still showing a reasonable range (200–400μs in the figure, 140–260μs for IPv4 forwarding).

Since our packet generator is not a hardware-based measuring instrument, the measurement result has two limitations: (*i*) measured latency numbers include delays incurred by the generator itself, and (*ii*) the generator only supports up to 28 Gbps due to overheads of measurement and rate limiting.

7. DISCUSSION

So far we have demonstrated that GPUs effectively bring extra processing capacity to software routers. We discuss related issues, the current limitations, and future directions here.

Integration with a control plane: Our immediate next step is to integrate control plane. We believe it is relatively straightforward to integrate Zebra [6] or Quagga [11] into PacketShader as other software routers. One issue arising here is how to update forwarding table in GPU memory without disturbing the data-path performance. This problem is not specific only to PacketShader, but also relevant to FIB update in traditional hardware-based routers. Incremental update or double buffering could be possible solutions.

Multi-functional, modular programming environment: PacketShader currently limits one GPU kernel function execution at a time per device. The multi-functionality support (e.g., IPv4 and IPsec at the same time) in PacketShader enforces to implement all the functions in a single GPU kernel. NVIDIA has recently added native support for concurrent execution of heterogeneous kernels into GTX480 [8], and we plan to modify the PacketShader framework to benefit from this feature. GTX480 also supports C++ and function pointers in kernel. We believe it will expedite our effort to implement a Click-like modular programming environment [30] in PacketShader.

Vertical scaling: As an alternative to using GPUs, it is possible to have more CPUs to scale up the single-box performance. However, having more CPUs in a single machine is not cost-effective due to upgrade of the motherboard, additional memory installation, and diminishing price/performance ratio. The CPU price per gigahertz in a single-socket machine[6] is $23 at present. The price goes up with more CPU sockets: $87 in a dual-socket machine[7] and $183 in a quad-socket machine[8].

In contrast, installing a cheap GPU (the price ranges from $50 to $500) into a free PCIe slot does not require extra cost. In case of limited PCIe slot space, one may add multiple GPUs into a slot with a PCIe switch. Although this approach would make the GPUs share the PCIe bandwidth, it can be useful for less bandwidth-hungry applications (e.g., header-only packet processing like IP forwarding).

Horizontal scaling: We have mostly focused on a single-machine router in this work. Our prototype router supports 8×10 GbE ports near 40 Gbps packet forwarding capacity for 64B IPv4 and IPv6 packets. In case more capacity or a larger number of ports are needed, we can take a similar approach as suggested by RouteBricks and use Valiant Load Balancing (VLB) [52] or direct VLB [19].

Power efficiency: In general, modern graphics cards (especially for high-end models) require more power than x86 CPU counterparts. The PacketShader server consumes 594W with two GTX480 cards and 353W without graphics cards under full load, showing 68% increase. The gap gets closer in an idle state: 327W and 260W, respectively. We believe that the increased power consumption is tolerable, considering the performance improvement from GPUs.

Opportunistic offloading: Our latency measurement results in Section 6.4 show that GPU acceleration incurs higher delay than CPU at low traffic. To address this issue, we have implemented the concept of *opportunistic offloading*, using CPU for low latency under light load and exploiting GPU for high throughput when heavily loaded, in our SSL acceleration project [27]. We plan to adopt this idea also in PacketShader for future work.

8. RELATED WORK

Today's multi-core network processors (NPs), combined with hardware multi-threading support, effectively hide memory access latency to handle large volume of traffic. For example, Cisco's QuantumFlow processors have 40 packet process engines, capable of handling up to 160 threads in parallel [4]. Similarly, Cavium Networks OCTEON II processors have 32 cores [2], and Intel IXP2850 processors have 16 microengines supporting 128 threads in total [46, 51]. We have realized the latency-hiding idea on commodity hardware, by exploiting massively-parallel capability of GPUs. GPUs also bring high computation power with full programmability, while NPs typically rely on hard-wired blocks to handle compute-intensive tasks, such as encryption and pattern matching.

Until recently, shared memory bus was the performance bottleneck of software routers. Bolla *et al*. report that the forwarding performance does not scale to the number of CPU cores due to FSB clogging [15]. FSB clogging happens due to cache coherency snoops in the multi-core architecture [54]. More recently, RouteBricks reports 2-3x performance improvement by eliminating FSB. It shows that a PC-based router can forward packets near 10 Gbps per machine. Moreover, RouteBricks breaks away from a single-box approach and scales the aggregate performance by clustering multiple machines [19]. In contrast, PacketShader utilizes GPU as a cheap source of computation capability and shows more than a factor of four performance improvement over RouteBricks on similar hardware. PacketShader could replace RB4, a cluster of four RouteBricks machines, with a single machine with better performance.

As a ubiquitous and cost-effective solution, GPUs have been widely used not only for graphics rendering but also for scientific and data-intensive workloads [5, 41]. Recently, GPUs have shown a substantial performance boost to many network-related workloads, including pattern matching [35, 48, 53], network coding [47], IP table lookup [35], and cryptography [24, 32, 49]. In our work, we have explored a general GPU acceleration framework as a complete system.

[6] $240 per Core i7 920 processor (2.66 GHz, 4 cores)
[7] $925 per Xeon X5550 processor (2.66 GHz, 4 cores)
[8] $2190 per Xeon E7540 processor (2.00 GHz, 6 cores)

9. CONCLUSIONS

We have presented PacketShader, a novel framework for high-performance network packet processing on commodity hardware. We minimize per-packet processing overhead in network stack and perform packet processing in the user space without serious performance penalty. On top of our optimized packet I/O engine, PacketShader brings GPUs to offload computation and memory-intensive workloads, exploiting the massively-parallel processing capability of GPU for high-performance packet processing. Careful design choices make PacketShader to be highly scalable with multi-core CPUs, high-speed NICs, and GPUs in NUMA systems. We have demonstrated the effectiveness of our approach with IPv4 and IPv6 forwarding, OpenFlow switching, and IPsec tunneling.

In this work we have shown that a well-designed PC-based router can achieve 40 Gbps forwarding performance with full programmability even on today's commodity hardware. We believe PacketShader will serve as a useful platform for scalable software routers on commodity hardware.

10. ACKNOWLEDGEMENT

We thank Katerina Arguraki, Vivek Pai, Seungyeop Han, anonymous reviewers, and our shepherd Robert Morris for their help and invaluable comments. This research was funded by KAIST High Risk High Return Project (HRHRP), NAP of Korea Research Council of Fundamental Science & Technology, and MKE (Ministry of Knowledge Economy of Repbulic of Korea, project no. N02100053).

11. REFERENCES

[1] AMD Fusion. http://fusion.amd.com.

[2] Cavium Networks OCTEON II processors. http://www.caviumnetworks.com/OCTEON_II_MIPS64.html.

[3] Check Point IP Security Appliances. http://www.checkpoint.com/products/ip-appliances/index.html.

[4] Cisco QuantumFlow Processors. http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.html.

[5] General Purpose computation on GPUs. http://www.gpgpu.org.

[6] GNU Zebra project. http://www.zebra.org.

[7] NVIDIA CUDA GPU Computing Discussion Forum. http://forums.nvidia.com/index.php?showtopic=104243.

[8] NVIDIA Fermi Architecture. http://www.nvidia.com/object/fermi_architecture.html.

[9] OpenFlow Reference System. http://www.openflowswitch.org/wp/downloads/.

[10] OpenFlow Switch Specification, Version 0.8.9. http://www.openflowswitch.org/documents/openflow-spec-v0.8.9.pdf.

[11] Quagga project. http://www.quagga.net.

[12] Receive-Side Scaling Enhancements in Windows Server 2008. http://www.microsoft.com/whdc/device/network/ndis_rss.mspx.

[13] The OpenFlow Switch Consortium. http://www.openflowswitch.org.

[14] University of Oregon RouteViews project. http://www.routeviews.org/.

[15] R. Bolla and R. Bruschi. PC-based software routers: High performance and application service support. In *ACM PRESTO*, 2008.

[16] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *USENIX Summer Technical Conference*, 1994.

[17] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *OSDI*, 2008.

[18] T. Brecht, G. J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner. Evaluating network processing efficiency with processor partitioning and asynchronous i/o. *SIGOPS Oper. Syst. Rev.*, 40(4):265–278, 2006.

[19] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *SOSP*, 2009.

[20] K. Fatahalian and M. Houston. A closer look at GPUs. *Communications of the ACM*, 51:50–57, 2008.

[21] A. Foong, J. Fung, and D. Newell. An in-depth analysis of the impact of processor affinity on network performance. In *IEEE ICON*, 2004.

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

[23] S. Han, K. Jang, K. Park, and S. Moon. Building a single-box 100 gbps software router. In *IEEE Workshop on Local and Metropolitan Area Networks*, 2010.

[24] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security*, 2008.

[25] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[26] V. Jacobson, C. Leres, and S. McCanne. libpcap, Lawrence Berkeley Laboratory, Berkeley, CA. http://www.tcpdump.org.

[27] K. Jang, S. Han, S. Moon, and K. Park. Converting your graphics card into high-performance SSL accelerator. *submitted for publication*.

[28] G. Jin and B. L. Tierney. System capability effects on algorithms for network bandwidth measurement. In *IMC*, 2003.

[29] D. Kim, J. Heo, J. Huh, J. Kim, and S. Yoon. HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs. In *Computer Graphics Forum*, volume 28, pages 1791–1800. John Wiley & Sons, 2009.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM TOCS*, 18(3):263–297, 2000.

[31] Y. Liao, D. Yin, and L. Gao. PdP: parallelizing data plane in virtual network substrate. In *ACM VISA*, 2009.

[32] S. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *IEEE Signal Processing and Communications*, 2007.

[33] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[34] J. Mogul and K. Ramarkishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM TOCS*, 15(3):217–252, 1997.

[35] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang. Ip routing processing with graphic processors. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010.

[36] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *ANCS*, 2008.

[37] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[38] NVIDIA Corporation. *NVIDIA CUDA Best Practices Guide, Version 3.0*.

[39] NVIDIA Corporation. NVIDIA CUDA Architecture Introduction and Overview, 2009.

[40] NVIDIA Corporation. NVIDIA CUDA Programming Guide, Version 3.0, 2009.

[41] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[42] K. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃijger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26:80–113, 2007.

[43] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *ACM PPoPP*, 2008.

[44] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Annual Linux Showcase & Conference*, 2001.

[45] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH*, 2008.

[46] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. Np-click: A productive software development approach for network processors. *IEEE Micro*, 24(5):45–54, 2004.

[47] H. Shojania, B. Li, and X. Wang. Nuclei: GPU-accelerated many-core network coding. In *IEEE INFOCOM*, 2009.

[48] R. Smith, N. Goyal, J. Ormont, C. Estan, and K. Sankaralingam. Evaluating GPUs for network packet signature matching. In *IEEE ISPASS*, 2009.

[49] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. *Cryptographic Hardware and Embedded Systems*, pages 79–99, 2008.

[50] J. Torrellas, H. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Trans. on Computers*, 43(6):651–663, 1994.

[51] J. S. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar. Supercharging planetlab: a high performance, multi-application, overlay network platform. *SIGCOMM CCR*, 37(4):85–96, 2007.

[52] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the ACM symposium on Theory of computing*, 1981.

[53] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, 2008.

[54] B. Veal and A. Foong. Performance Scalability of a Multi-Core Web Server. In *ANCS*, 2007.

[55] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *SIGCOMM*, 1997.