Packet Processing on the GPU

Matthew Mukerjee
Carnegie Mellon University
mukerjee@cs.cmu.edu

David Naylor
Carnegie Mellon University
dnaylor@cs.cmu.edu

Bruno Vavala
Carnegie Mellon University
bvavala@cs.cmu.edu

ABSTRACT

Packet processing in routers is traditionally implemented in hardware; specialized ASICs are employed to forward packets at line rates of up to 100 Gbps. Recently, however, improving hardware and an interest in complex packet processing has prompted the networking community to explore software routers; though they are more flexible and easier to program, achieving forwarding rates comparable to hardware routers is challenging.

Given the highly parallel nature of packet processing, a GPU-based router architecture is an inviting option (and one which the community has begun to explore). In this project, we explore the pitfalls and payoffs of implementing a GPU-based software router.

1. INTRODUCTION

There are two approaches to implementing packet processing functionality in routers: bake the processing logic into hardware, or write it in software. The advantage of the former is speed; today's fastest routers perform forwarding lookups in dedicated ASICs and achieve line rates of up to 100 Gbps (the current standard line rate for a core router is 40 Gbps) [2]. On the other hand, the appeal of software routers is programability; software routers can more easily support complicated packet processing functions and can be reprogrammed with updated functionality.

Traditionally, router manufacturers have opted for hardware designs since software routers could not come close to achieving the same forwarding rates. Recently, however, software routers have begun gaining attention in the networking community for two primary reasons:

1. Commodity hardware has improved significantly, making it possible for software routers to achieve line rates. For example, PacketShader forwards packets at 40 Gbps [2].

2. Researchers are introducing increasingly more complex packet processing functionality. A good example of this trend is Software Defined Networking [5], wherein each packet is classified as belonging to one or more *flows* and is forwarded accordingly; moreover, the *flow rules* governing forwarding behavior can be updated on the order of minutes [5] or even seconds [3].

Part of this new-found attention for software routers has been an exploration of various hardware architectures that might be best suited for supporting software-based packet processing. Since packet processing is naturally an SIMD application, a GPU-based router is a promising candidate. The primary job of a router is to decide, based on a packet's destination address, through which output port the packet should be sent. Since modern GPUs contain hundreds of cores [8], they could potentially perform this lookup on hundreds of packets in parallel. The same goes for more complex packet processing functions, like flow rule matching in a software defined network (SDN).

This project is motivated by two goals: first, we seek to implement our own GPU-based software router and compare our experience to that described by the PacketShader team (see §2). Given time (and budget!) constraints, it is not feasible to expect our system to surpass PacketShader's performance. Our second goal is to explore scheduling the execution of multiple packet processing functions (e.g., longest prefix match and firewall rule matching).

The rest of this paper is organized as follows: first we review related work that uses the GPU for network processing in §2. Then we explain the design of our system in §3. §4 describes our evaluation setup and presents the results of multiple iterations of our router. Finally, we discuss our system and future work in §5 and conclude in §6.

2. RELATED WORK

We are by no means the first to explore the use of a GPU for packet processing. We briefly summarize here the contributions of some notable projects.

PacketShader [2] implements IPv4/IPv6 forwarding, IPsec encryption, and OpenFlow [5] flow matching on the GPU using NVIDIA's CUDA architecture. They were the first to demonstrate the feasability of multi-10Gbps software routers.

A second key contribution from PacketShader is a highly optimized packet I/O engine. By allocating memory in the NIC driver for batches of packets at a time rather than individually and by removing unneeded meta-data fields (parts of the Linux networking stack needed by endhosts are never used in a software router), they achieve much higher forwarding speeds, even before incorporating the GPU. Due to time constraints, we do not make equivalent optimizations in our NIC driver, and so we cannot directly compare our results with PacketShader's.

Gnort [11] ports the Snort intrusion detection system (IDS) to the GPU (also using CUDA). Gnort uses the same basic CPU/GPU workflow introduced by PacketShader (see §3); its primary contribution is implementing fast string pattern-matching on the GPU.

Hermes [12] builds on PacketShader by implementing a CPU/GPU software router that dynamically adjusts batch size to simultaneously optimize multiple QoS metrics (e.g., bandwidth and latency).

3. BASIC SYSTEM DESIGN

We use the same system design presented by PacketShader and Gnort. Pictured in Figure 1, packets pass through our system as follows: (1) As packets arrive at the NIC, they are copied to main memory via DMA. (2) Software running on the CPU copies these new packets to a buffer until a sufficient number have arrived (see below) and (3) the batch is transferred to memory on the GPU. (4) The GPU processes the packets in parallel and fills a buffer of results on the GPU which (5) the CPU copies back to main memory when processing has finished. (6) Using these results, the CPU instructs the NIC(s) where to forward each packet in the batch; (7) finally, the NIC(s) fetch(es) the packets from main memory with another DMA and forwards them.

3.1 GPU Programming Issues

Pipelining. As with almost any CUDA program, ours employs pipelining for data transfers between main memory and GPU memory. While the GPU is busy processing a batch of packets, we can utilize the CPU to copy the results from the previous

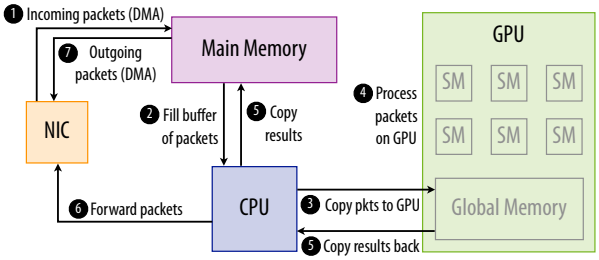


Figure 1: Basic System Design

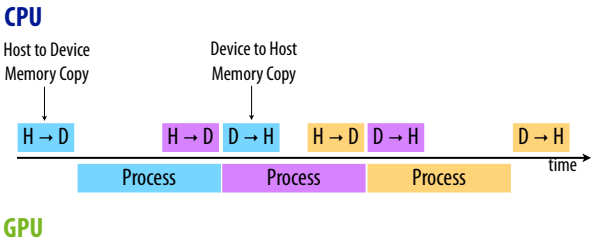


Figure 2: Pipelined Execution

batch of packets from the GPU back to main memory and copy the next batch of packets to the GPU (Figure 2). In this way, we never let CPU cycles go idle while the GPU is working.

Batching. Although the GPU can process hundreds of packets in parallel (unlike the CPU), there is, of course, a cost: non-trivial overhead is incurred in transferring packets to the GPU and copying results back from it. To amortize this cost, we process packets in batches. As packets arrive at our router, we buffer them until we have a batch of some fixed size `BATCH_SIZE`, at which point we copy the whole batch to the GPU for processing.

Though batching improves throughput, it can have an adverse effect on latency. The first packet of a batch arrives at the router and is buffered until the remaining `BATCH_SIZE`-1 packets arrive rather than being processed right away. To ensure that no packet suffers unbounded latency (i.e., in case there is a lull in traffic and the batch doesn't fill), we introduce another parameter: `MAX_WAIT`. If a batch doesn't fill after `MAX_WAIT` milliseconds, the partial batch is transferred to GPU for processing. We evaluate this tradeoff in §4.

Mapped Memory. Newer GPUs (such as ours) have the ability to directly access host memory that has been *pinned and mapped*, eliminating steps (3) and (5). Though data clearly still needs to be copied to the GPU and back, using mapped memory rather than explicitly performing the entire copy at once allows CUDA to copy individual lines "intelligently"

as needed, overlapping copies with execution. We evaluate the impact of using mapped memory in §4 as well.

3.2 Packet Processing

A router performs one or more packet processing functions on each packet if forwards. This includes deciding where to forward the packet (via a longest prefix match lookup on the destination address or matching against a set of flow rules) and might also include deciding whether or not to drop the packet (by comparing the packet header against a set of firewall rules or comparing the header and payload to a set of intrusion detection system rules). In our system we implement two packet processing functions: longest prefix match and firewall rule matching. We pick longest prefix match, because it is the core algorithm used by routers to determine which port to forward a packet on. We additionally choose firewall rule matching as it is a very common feature on commodity routers that is simple to implement (only requiring packet header inspection).

3.2.1 Longest Prefix Match

Long Prefix Match is an algorithm for route lookup. It performs a lookup on a data structure, whose entries are prefix-port pairs. The prefixes identify routes and have the form *IP address/mask* (e.g., 192.168.1.0/24). The ports represent the link to reach the correct the next hop, possibly the destination. Once the router receives a packet to forward, it selects an entry in the data structure such that the prefix is the longest (i.e., with the largest mask) available that matches the destination address in the packet. Then, the associated output port is returned.

We adapted the algorithm from [9] for our work. The algorithm uses a *binary trie*. It is a tree-like data structure that keeps the prefixes ordered for fast lookup. In particular, the position of each internal node in the trie defines a prefix which is constant in the subtree associated to such node. The depth of a prefix is the length of its mask (i.e., between 0 and 32, in our case), which is thus a bound for the lookup time.

The lookup procedure is extremely simple. A depth-first search in run on the trie using the destination address of the packet being processed. So each node lying on the path describing the associated prefix is visited. At each step, it updates the output ports, if an entry is stored (i.e., longest match). Clearly, the last update is returned, possibly none as there may be no matching prefixes.

3.2.2 Firewall Rule Matching

A firewall rule is defined by a set of five values (the "5-tuple"): source and destination IP address, source and destination port number, and protocol. The protocol identifies who should process the packet after IP (e.g., TCP, UDP, or ICMP). Since most traffic is either TCP or UDP [10], the source and destination port numbers are also part of the 5-tuple even though they are not part of the network-layer header. If the protocol is ICMP, these fields can instead be used to encode the ICMP message type and code.

Each rule is also associated with an action (usually ACCEPT or REJECT). If a packet matches a rule (that is, the values in the packet's 5-tuple match those in the rule's 5-tuple), the corresponding action is performed. Rather than specifying a particular value, a rule might define a range of matching values for one of the fields of its 5-tuple. A rule might also use a wildcard (*) for one of the values if that field should not be considered when deciding if a packet matches. For example, a corporate firewall might include the following rule: (*, [webserver-IP], *, 80, TCP):ALLOW. This rule allows incoming traffic from any IP address or port to access the company's webserver using TCP on port 80.

Our rule matching algorithm is incredibly simple. Rules are stored in order of priority and a packet is checked against each one after the next in a linear search. Though this sounds like a naïve approach, [7] claims that it is the approach taken by open source firewalls like `pf` and `iptables` and likely other commercial firewalls as well. We discuss how we generate the rules used in our tests in §4.1.4.

4. EVALUATION

4.1 Experimental Setup

4.1.1 Hardware

We ran our tests on a mid-2012 MacBook Pro with a 2.6 GHz Core i7 processor, 8GB RAM, and an NVIDIA GeForce GT 650M graphics card with 1GB memory.

4.1.2 Router Framework

We implemented a software router framework capable of running in one of two modes: CPU-only or CPU/GPU. We use the router in CPU-only mode to gather a baseline against which we can compare the performance of CPU/GPU mode. In either mode, the framework handles gathering batches of packets, passing them to a "pluggable" packet processing

function, and forwarding them based on the results of processing.

We implemented three processing functions: one that does longest prefix match, one that does firewall rule matching, and one that does both. For each we implemented CPU version and a GPU version (the GPU version is a CUDA kernel function).

4.1.3 Packet Generation

We use the Click Modular Router [4] to generate packets for our software router framework to process. We modify the standard packet generation functions in click to output UDP packets with random source and destination addresses as well as random source and destination ports. We have click generate these randomly addressed packets as fast as it can and have it send the packets up to our software routing framework via a standard Berkeley socket between both locally hosted applications.

Note that this "full-on" kind of workload is essential to test the maximum throughput of our router framework, but does not necessarily model a particular kind of real-world workload. However, in the case of our system, we wish to measure the feasibility of processing packets in software at the same speed as specially-designed hardware ASICs. Thus our evaluation focuses on comparisons of maximum throughput.

4.1.4 Packet Processing

Longest Prefix Match. First of all, since CUDA provides limited support for dynamic data structures such as a trie, we adapted the algorithm to run on the GPU. In particular, in the initial setup of the trie, we serialize the data structure into an array, so that it can be easily transferred on the GPU's memory.

In order to work with a realistic dataset, we initialized the FIB with the prefixes in the Internet belonging to actual Autonomous Systems. The list has been retrieved from CAIDA [1], which periodically stores a simplified version of its RouteViews Collectors' BGP tables. Overall, the size of the trie turns out to be a few tens of megabytes large. Once built, the serialized trie is transferred to the GPU to be used during lookup.

Firewall Rule Matching. To evaluate the performance of our router's firewall rule matching function, we generate a set of random firewall rules. The 5-tuple values for our random rules are chosen according to the distributions in [7]. For example, we use the probabilities in Table 1, taken from [7], to pick a new random rule's protocol. Similar statistics were used to pick source/destination address/port.

Protocol	Prevelance in Rules
TCP	75%
UDP	14%
ICMP	4%
*	6%
Other	1%

Table 1: Protocol distribution in firewall rules

4.2 Evaluation Metrics

We use three different metrics to evaluate our router's performance. The results in §4.3 present these values averaged over 30 seconds of router operation.

Bandwidth. No router performance evaluation would be complete without considering bandwidth, and so we measure the number of 64 byte packets forwarded by our router per second, from which we calculate bandwidth in gigabits per second. For comparison, the norm for core routers is about 40 Gbps with high-end routers currently maxing out around 100 Gbps.

Latency. Bandwidth only tells part of the story, however; the delay a single packet experiences at a router is important as well (for some applications, it is more important than bandwidth). Since some optimizations aimed at increasing our router's bandwidth increase latency as well (such as increasing batch sizes), measuring latency is important.

We measure both the minimum and maximum latencies of our router. The maximum latency is the time from the arrivial of the first packet of a batch to the time it is forwarded; the minimum latency is the same but for the last packet of a batch.

Processing Time. We also consider a microbenchmark: the time spent in the packet processing function itself (i.e., doing the longest prefix match lookup or matching against firewall rules). This is largely a sanity check; we expect to see the GPU perform much better here, though actual performance (measured by bandwidth and latency) depends on many other factors (like the time spent copying data to/from the GPU).

4.3 Results

Our router went through four iterations, each one introducing optimizations based on what we learned from the results of the last. We therefore present our results in four stages, guiding the reader through our design process.

4.3.1 Iteration One

We began by naïvely implementing the design presented in §3; at this point, we made no optimizations

— the goal was building a functioning GPU-based router.

Not surprisingly, it performed underwhelmingly. The GPU version of our router achieved roughly 80% of the bandwidth the CPU version did (Figure 3(a)) and its (max and min) latencies were 1.3X longer (Figure 3(b)). The processing time is not to blame; as expected, the GPU performs the actual packet processing much faster than the CPU (Figure 3(c) — the GPU processing time is so small it hugs the X axis). A quick examination of the time spent performing different functions (Figure 3(d)) explains where the GPU router is losing ground: although it spends less time processing, it has to copy packets to the GPU for processing and copy the results back, tasks the CPU version doesn't need to worry about (Figure 3(e)).

4.3.2 Iteration Two

Since both of our packet processing functions operate only on the data carried by the packet header, we can reduce the time spent copying data to the GPU by copying only the packet headers. Unfortunately, the results do not contain unnecessary information, and cannot easily be condensed (this is not completely true — it is probably possible to compress the results, though we do not explore this here).

Figure 4 shows the performance of the second iteration of our router. Reducing the number of bytes copied to the GPU has closed the gap between the CPU-only and the CPU/GPU routers in terms of bandwidth and latency, but the CPU/GPU router still doesn't perform any better. Even though we have all but eliminated copy time to GPU, Figure 4(c) suggests that we should try to cut down copy time from the GPU as well.

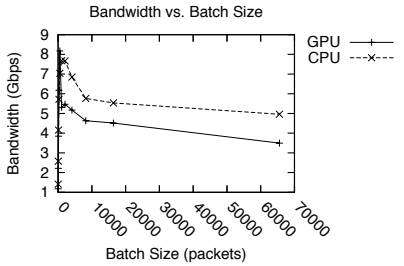
4.3.3 Iteration Three

Unfortunately, there is no unnecessary data being copied back from the GPU as there was being copied to it; we must find another way to reduce the copy-from-GPU overhead. Instead, we modify our router's workflow to take advantage of mapped memory (§3.1).

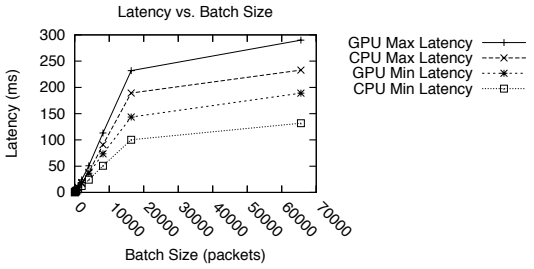
This gives the CPU/GPU router a tiny edge over the CPU-only router (Figure 5), but the gains are small (as Amdahl's Law would suggest — the copy-from-device time we eliminated was a small portion of the total time in Figure 4(c)).

4.3.4 Iteration Four

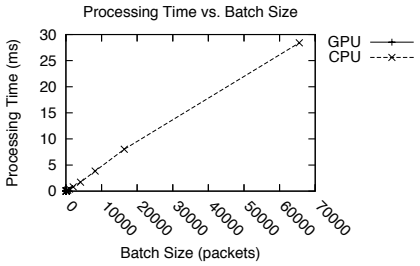
Having eliminated the overhead of copying data to and from the GPU, the third iteration of our



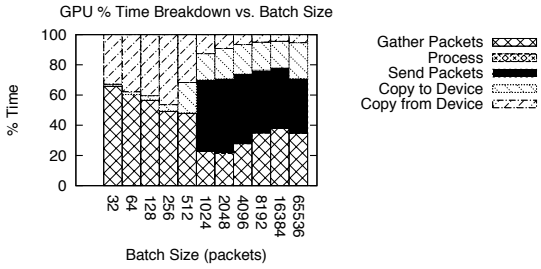
(a) Bandwidth vs. Batch Size



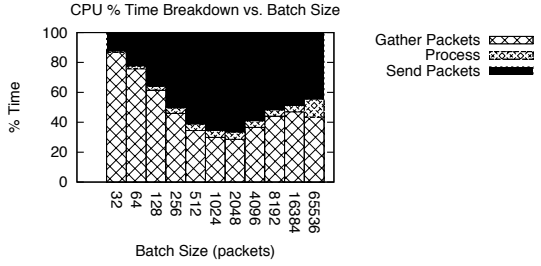
(b) Latency vs. Batch Size



(c) Processing Time vs. Batch Size

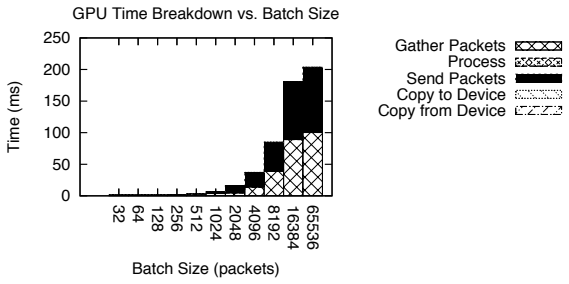


(d) Breakdown of Relative GPU Time

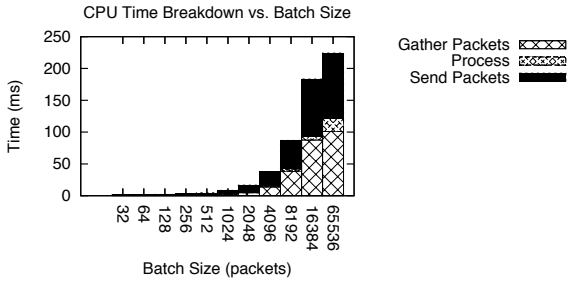


(e) Breakdown of Relative CPU Time

Figure 3: Iteration 1 Results

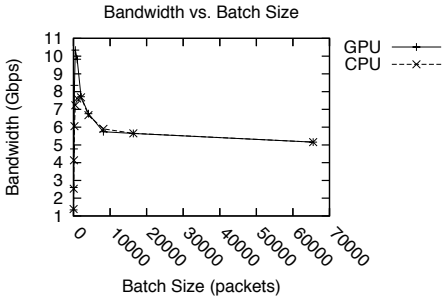


(a) Breakdown of Absolute GPU Time

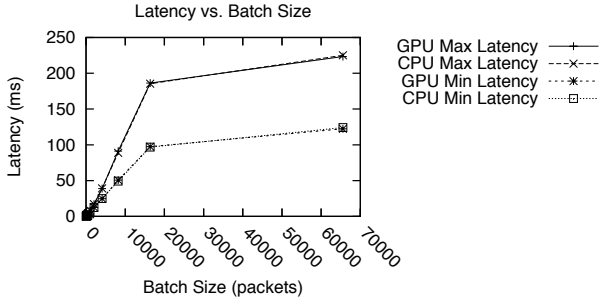


(b) Breakdown of Absolute CPU Time

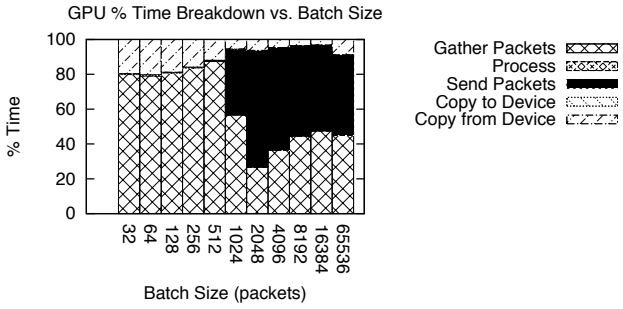
Figure 6: Iteration 3 Breakdown



(a) Bandwidth vs. Batch Size

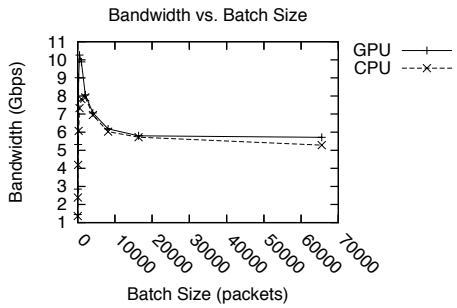


(b) Latency vs. Batch Size



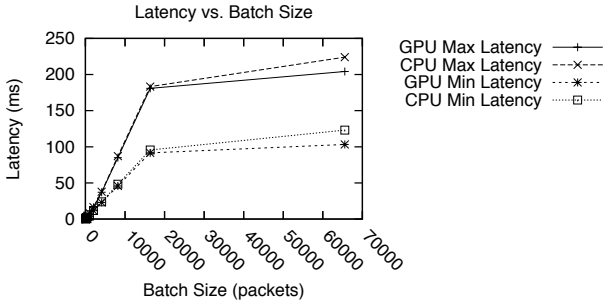
(c) Breakdown of Relative GPU Time

Figure 4: Iteration 2 Results

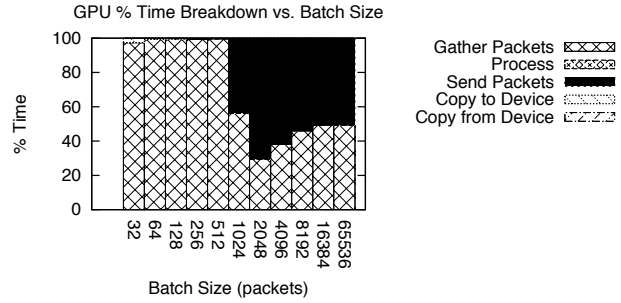


e

(a) Bandwidth vs. Batch Size



(b) Latency vs. Batch Size



(c) Breakdown of Relative GPU Time

Figure 5: Iteration 3 Results

CPU/GPU router only displays meager gains over the CPU-only version. We turn to a breakdown by function of the absolute runtime of each (Figure 6) to understand why.

The CPU-only router clearly spends more time in the processing function; however, both spend nearly all their time performing packet I/O (that is, receiving packets from Click and forwarding them after processing), so the difference is processing times has little effect. This suggests that our Click-based packet generator is the bottleneck. (Indeed, PacketShader spent a great amount of time developing highly optimized drivers for their NIC [2].) To test this hypothesis, we implemented a version of our packet generator that pre-generates a buffer of random packets and returns this buffer immediately when queried for a new batch; similarly, when results are returned so that packets may be forwarded, the call returns immediately rather than waiting for the whole batch to be sent.

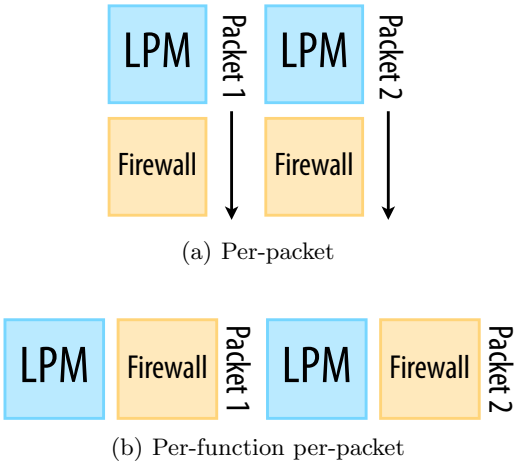
Sure enough, the CPU/GPU router now operates at 3X the bandwidth of the CPU-only version and with 1/5th the latency (Figure 7). Of course, the bandwidth achieved by both is completely unrealistic (instantaneous packet I/O is impossible), but these results indicate that with optimized packet I/O drivers like PacketShader's, our CPU/GPU router would indeed outperform our CPU-only one.

5. DISCUSSION AND FUTURE WORK

Multithreaded CPU Processing. The comparison of our CPU/GPU router to the CPU-only baseline is not completely fair. Our CPU-only router operates in a single thread, yielding misleadingly low performance. Any CPU-based software router in the real world would certainly spread packet processing across multiple cores, and we would be surprised if any core software router were run on a machine with fewer than 16 cores.

A simple extension of our project would be to parallelize our CPU-only packet processing functions with OpenMP to provide more realistic baseline measurements.

We have implemented a simple multi-threaded version. Our preliminary results show that OpenMP can dramatically improve the performance. By tuning the number of threads and the batch size, we could increase the bandwidth up to one order of magnitude, with respect to the mono-thread version. However, the packet processing on the GPU still outperforms that on the CPU, being around 20% faster. More investigation is however required to understand the impact of the parameters on the performance.



(a) Per-packet

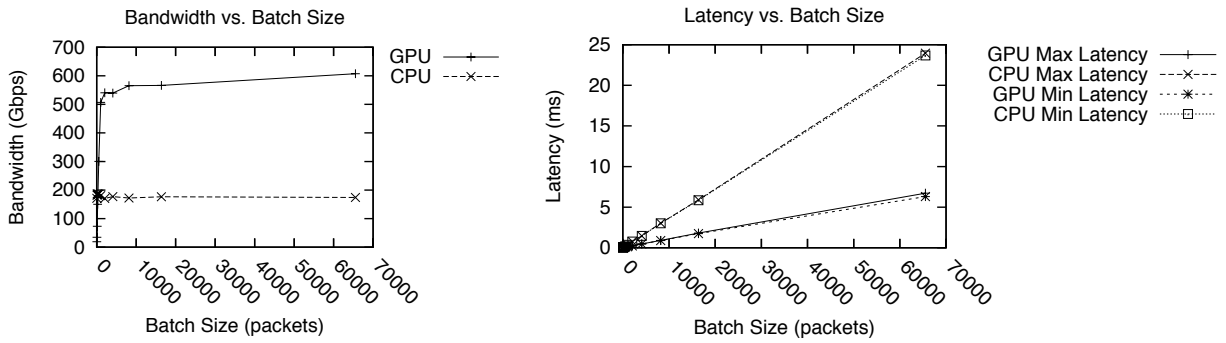
(b) Per-function per-packet

Figure 8: Scheduling Multiple Processing Functions

Harder Processing Functions. Even in the final iteration of our router, the CPU/GPU version only achieves slightly more than three times the bandwidth of the CPU-only version. Though this is by no means an improvement to scoff at, the speedup strikes us as being a tad low. We suspect the cause is that our packet processing functions are not taxing enough; the harder the processing function, the more benefit we should see from the massively parallel GPU. This suggests that GPU-based software routers might be best suited for complex packet processing like IDS filtering (which requires pattern matching against packet payloads) and IPsec processing (which requires expensive cryptographic operations).

One of our goals was to explore the best way to schedule multiple packet processing functions on the GPU. As a simple example, to schedule both LPM and firewall lookup, you could imagine parallelizing per-packet or parallelizing per-function (Figure 8). We began testing these schemes, but, as our processing functions turned out to be "easy," there was little difference. We think this is still an interesting question to explore with more taxing processing functions.

Faster Packet I/O. By far the largest issue we noted is that generating and gathering packets from Click contributes to the majority of the latency, becoming the dominant part of both CPU and GPU overall latency. As seen in Figure 7, removing the overhead contributed by generating and gather packets, our GPU implementation performs much better than our CPU implementation. This is the same conclusion that the authors of PacketShader [2] came to. Thus in their system, they focused on reimplementing the driver for their NIC to alleviate these issues. In our framework we could similarly emu-



(a) Bandwidth vs. Batch Size

(b) Latency vs. Batch Size

Figure 7: Iteration 4 Results

late newer NIC technologies (like RDAM) to allow zero-copy access to DRAM that is memory mapped in the GPU. We could emulate this by having Click copy packets directly to application DRAM rather than first sending the packets to the application via a socket.

Streams. The typical use of a GPU involves the following operations: a set of tasks and input data is first collected, transferred to the GPU, processed, and eventually the output is copied back to the host device. These operations are performed cyclically, often on independent data, as in the case of our packet processing applications. Since the CUDA and host device are separate processing unit, it is critical for performance to maximize the concurrency of such operations.

CUDA Streams provide an interesting mechanism to optimize GPU-based applications. Basically, CUDA allows to assign operations (like kernel execution and *asynchronous* data transfer to and from the device) to streams. Then, the stream executions are overlapped in such a way that similar operations of different streams do not interfere with each other (i.e., a kernel execution in stream1 and a data transfer in stream2 can be performed in parallel). The high level idea is quite similar to that of instruction pipelining.

We have implemented a version of our router using CUDA streams. Our preliminary results show that streams indeed increase the parallelism and throughput. Operation overlapping is quite visible in the Visual Profiler. The number of processed packets increases substantially (around 30-40%). However, more investigation is required on this subject, particularly on the packet generator and on the new issues related to resource management, highlighted by the NVidia Visual Profiler.

Integrated Graphics Processors. One tempting idea to explore is the use of integrated graphics processors rather than dedicated discrete GPUs. Modern processors (Intel Core i-series, etc.) include a traditional multi-core GPU directly on the processor itself. In essence, this shifts the position of the GPU from being on the PCI-express bus to being co-located with the CPU on the quickpath interconnect (QPI). As the QPI can potentially provide more bus bandwidth to memory, and integrated graphics processor could obtain even higher maximum throughput, as memory constraints are the biggest source of potential slowdown after packet I/O at the NIC.

6. CONCLUSION

Software routers are no doubt poised to start playing a larger and larger role in real systems. The popularity of software defined networks like OpenFlow alone is a big enough driver to ensure this, though software routers might find other niches as well. To our knowledge, GPU-based software routers are currently limited to the research lab, and we are unsure whether this is likely to change. Our experience (and PacketShader's success) suggests that they might be a viable option, though the effort required to optimize them may in the end outweigh the benefits.

7. REFERENCES

[1] CAIDA. Route Views, http://data.caida.org/datasets/routing/routeviews-prefix2as/2012/01/, 2012.

[2] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[3] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 127–132, New York, NY, USA, 2012. ACM.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

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

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

[7] D. Rovniagin and A. Wool. The geometric efficient matching algorithm for firewalls. *IEEE Trans. Dependable Secur. Comput.*, 8(1):147–159, Jan. 2011.

[8] 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 *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[9] S. D. Strowes. LPM Trie, http://svn.sdstrowes.co.uk/pub/longest-prefix/, 2011.

[10] K. Thompson, G. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *Network, IEEE*, 11(6):10–23, 1997.

[11] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Y. Zhu, Y. Deng, and Y. Chen. Hermes: an integrated cpu/gpu microarchitecture for ip routing. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 1044–1049, New York, NY, USA, 2011. ACM.

APPENDIX

A. DISTRIBUTION OF CREDIT

Group Member	Portion of Credit
Matt	33.3%
David	33.4%
Bruno	33.3%