Fast and Flexible:
Parallel Packet Processing with GPUs and Click

Weibin Sun
Flux Research Group
University of Utah, School of Computing
Salt Lake City, UT, USA
wbsun@cs.utah.edu

Robert Ricci
Flux Research Group
University of Utah, School of Computing
Salt Lake City, UT, USA
ricci@cs.utah.edu

ABSTRACT

We introduce Snap, a framework for packet processing that outperforms traditional software routers by exploiting the parallelism available on modern GPUs. While obtaining high performance, it remains extremely flexible, with packet processing tasks implemented as simple modular elements that are composed to build fully functional routers and switches. Snap is based on the Click modular router, which it extends by adding new architectural features that support batched packet processing, memory structures optimized for offloading to coprocessors, and asynchronous scheduling with in-order completion. We show that Snap can run complex pipelines at high speeds on commodity PC hardware by building an IP router incorporating both an IDS-like full-packet string matcher and an SDN-like packet classifier. In this configuration, Snap is able to forward 40 million packets per second, saturating four 10 Gbps NICs at packet sizes as small as 128 byes. This represents an increase in throughput of nearly 4x over the baseline Click running comparable elements on the CPU.

1. INTRODUCTION

As networks advance, the need for high-performance packet processing in the network increases for two reasons: first, networks get faster, and second, we expect more functionality from them [28, 17, 10, 4, 8, 3]. The nature of packet data naturally lends itself to parallel processing [7], and recent work has demonstrated that GPUs, which have thousands of cores, are well-suited to a number of packet-processing tasks. These include, but are not limited to, route lookup [9], encryption [12], and deep packet inspection [25, 11]. However, a software router is made up of more than just these heavyweight processing and lookup elements. A range of other elements are needed to build a fully functional router, including "fast path" elements such as TTL decrement, checksum recalculation, and broadcast management, and "slow path" elements such as handling of IP options, ICMP, and ARP. Building new features not present in today's routers adds even more complexity. To take full advantage of the GPU in a packet processor, what is needed is a flexible, modular framework for building complete processing pipelines by composing GPU programs with each other and with CPU code.

We have designed and implemented Snap to address this need. It extends the architecture of the Click modular router [15] to support offloading parts of a packet processor onto the GPU. Snap enables individual elements, the building blocks of a Click processing pipeline, to be implemented as GPU code. It extends Click with "wide" ports that pass batches of packets, suitable for processing in parallel, between elements. Snap also provides elements that act as adapters between serial portions of the processing pipeline and parallel ones, handling the details of batching up packets, efficiently

copying between main memory and the GPU, scheduling GPU execution, and directing the outputs from elements into different paths on the processing pipeline. In addition to these user-visible changes to Click, Snap also makes a number of "under the hood" changes to the way that Click manages memory and optimizes its packet I/O mechanisms to support multi-10 Gbit rates.

This paper makes three contributions. First, we design a set of enhancements to the Click architecture that make it easy to integrate GPU elements into a processing pipeline. These extensions are in keeping with Click's design philosophy of using simple modular elements that can be composed to form a variety of router configurations. Second, we develop a set of techniques that enable a high-performance implementation of these extensions. These include packet slicing to save bus bandwidth, predicated execution to handle divergent paths through the element graph, and hybrid synchronous/asynchronous packet pipeline scheduling with in-order completion. Third, we implement a set of GPU-accelerated elements that provide common processing functionality, which serve as demonstrations of Snap's usefulness. We show that these elements achieve high rates of forwarding performance on a commodity PC, even when complex computational elements are used in combination. When combining an IP router, an SDN-like packet classifier, and an IDS-like full-payload string matcher, Snap forwards at 44 Mpps, reaching full line rate on four 10 Gbps NICs for all but the smallest packets. This yields a speedup of nearly 4x over the baseline CPU-based Click.

2. BACKGROUND

Before describing Snap's design and implementation, we begin with short introductions to GPU computing and Click. We present a simple set of experiments that demonstrate that introducing computation-heavy elements into Click configurations reduces performance, and the potential for the GPU to alleviate this problem. We also give a brief survey of related work.

2.1 GPU Computing Basics

A modern programmable GPU acts as a co-processor: it receives code (called "kernels"), data, and commands from the host CPU. The main focus of GPU evolution is increasing parallelism: current high-end GPUs have more than two thousands cores [19]. Because they are high-volume, mass-market devices, a large amount of engineering effort goes into constant improvements, and they are cheaper than more specialized parallel offload engines. A key challenge of GPU computing is that GPU code requires bulk data to see a speedup over CPU code. A single GPU core has much lower performance than a CPU core, but together, the thousands of cores in a GPU can provide greater throughput than the CPU for many types of highly parallel tasks. For maximum utilization, it is desirable to launch

many threads per GPU core: GPUs use zero-cost thread switching to hide memory access latency, and thus typically require at least tens of thousands of threads to achieve good utilization. It is not unusual for scientific computing workloads to launch millions of simultaneous threads. GPUs use a SIMT (Single Instruction, Multiple Thread) execution model, in which a group of cores shares a single program counter, executing threads in lockstep.

GPUs have their own on-board device memory, which can be as large as 6 GB [19]. GPU cores can only access the device memory directly, so data must be copied in from the PC's main memory (called "host memory" in GPU computing) via DMA over the PCIe bus. The PCIe bus is also used for CPU-GPU communication, such as launching GPU kernels and synchronizing states as the computation progresses. The result is that there is overhead every time a GPU kernel is launched, and managing this overhead is an important part of using the GPU as an offloading engine. Fortunately, bulk data copies and large thread counts can be used to amortize this overhead when the data being operated on is large enough. For analyses of these overheads, we refer the reader to PacketShader [9], SSLShader [12], and GPUstore [23]. More detail about GPU computing, especially the CUDA environment that we used to implement Snap, can be found in the CUDA Guide [18].

2.2 Click

Click is a modular software router that provides an efficient pipeline-like abstraction for packet processing on PC hardware. A packet processor is constructed by connecting small software modules called "elements" into a graph called a "configuration." Click elements have two kinds of "ports:" input ports and output ports, and a single element may have more than one of each. A connection between two elements is made by connecting an output port of one element to an input port of another. Packets move along these connections when they are pushed or pulled: an element at the head of the pipeline can push packets downstream, or packets can be pulled from upstream by elements at the tail of the pipeline. Packets typically enter Click at a `FromDevice` element, which receives them from a physical NIC and pushes them downstream as they arrive. Unless dropped, packets leave through a `ToDevice` element, which pulls them from upstream and transmits them as fast as the outgoing NIC allows. Queues are used to buffer packets between push and pull sections of the configuration.

At the C++ source-code level, elements are written as subclasses of a base `Element` class, ports are instances of a `Port` class, and network packets, which are represented by instances of the `Packet` class, are passed one at a time between `Element`s by calling the elements' push() or pull() methods. We run Click at user level—although Click can run directly in the kernel, with the Netmap [20] zero-copy packet I/O engine, user-level Click has a higher forwarding rate than the kernel version [20].

2.3 Motivating Experiments

Our work on Snap is motivated by two facts: (1) rich packet processing functionality can represent a major bottleneck in processing pipelines; and (2) by offloading that functionality onto a GPU, large performance improvements are possible, speeding up the entire pipeline. We demonstrate these facts with two motivating experiments. (Our experiment setup and methodologies are described in more detail in Section 4.)

Our first experiment starts with the simplest possible Click forwarder, shown at the top of Figure 1. It does no processing on packets. It simply forwards them from one interface to another. We then add, one at a time, elements that do IP route lookup, classification based on header bits (as is done in most SDN designs),
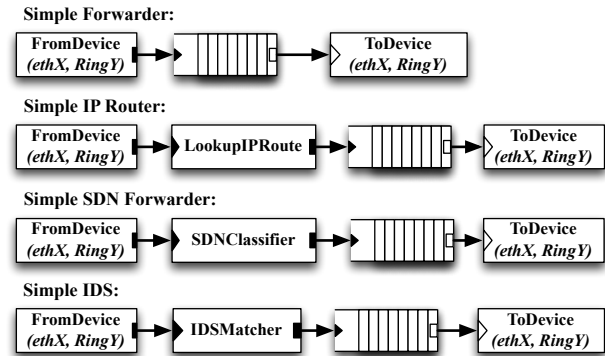

Figure 1: Simple Click configurations

Configuration	Throughput	Relative TPut
Simple Forwarder	30.97 Gbps	100%
IP Router	19.4 Gbps	62.7%
IDS	17.7 Gbps	57.3%
SDN Forwarder	18.8 Gbps	60.7%

Table 1: Relative throughputs of simple processing pipelines.

and string matching (used in many intrusion detection systems and deep-packet-inspection firewalls). The relative throughputs of these four configurations, normalized to the throughput of the Simple Forwarder, are compared in Table 1. We can clearly see from this table that the addition of a even a single, common, processing task into the forwarding pipeline can significantly impact performance, cutting throughput by as much as 43%. In short, processing does represent a bottleneck, and if we can speed it up, we can improve router throughput.

Our next set of experiments compares the performance of these three processing element when run on the CPU and on the GPU. These experiments involve no packet I/O—we are simply interested in discovering whether the raw performance of the GPU algorithms offers enough of a speedup to make offloading attractive. We process packets in batches, which is necessary to get parallel speedup on the GPU. The results are shown in Figure 2. Two things become clear from these graphs. First, GPUs do indeed offer impressive speedups for these tasks: we see a 16x speedup for IP route lookup (559 Mpps on the GPU vs 34.7 Mpps on the CPU). Second, fairly large batches of packets are needed to achieve this speedup. These results are in line with findings from earlier studies [25, 9].

GPUs are not appropriate for every type of packet processing element. In particular, elements that require a guarantee that they see every packet in a flow in order, or that have heavy state synchronization requirements, are not well-suited to massively parallel processing. Our challenge in Snap is to make it possible to take advantage of GPU parallelism in a practical a way that preserves the inherent composability and flexibility of Click, including incorporation of CPU elements into the processing pipeline.

2.4 Related Work

GPU-accelerated packet processing has begun to appear in the literature in the past few years [9, 12, 25, 11, 26]. This work focuses on offloading specific processing tasks to the GPU, such as route lookup, encryption, string matching, and name resolution. It uses specialized engines for offloading packets to the GPU; these engines, such as the `psio` engine created for PacketShader [9], are highly
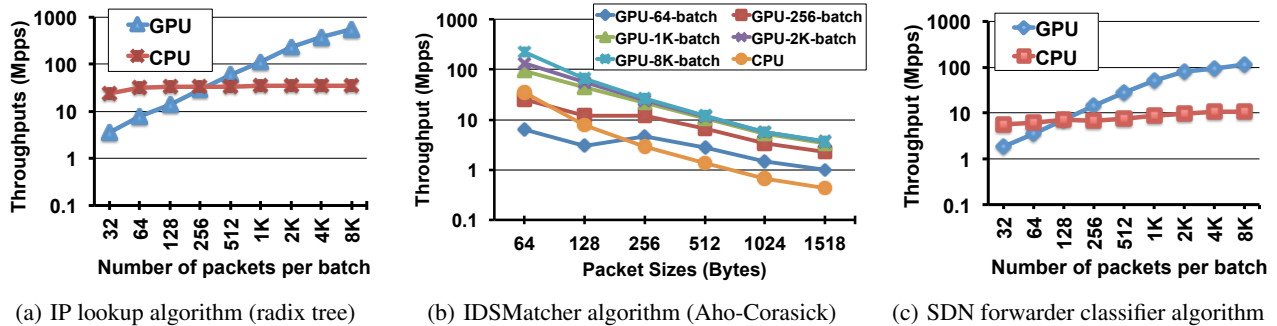
(a) IP lookup algorithm (radix tree)　　(b) IDSMatcher algorithm (Aho-Corasick)　　(c) SDN forwarder classifier algorithm

Figure 2: GPU vs. CPU performance on packet processing algorithms. Note that the axes are non-linear.

efficient, but they are not well suited to building pipelines more complex than a single processing function. In particular, they do not directly provide efficient support for splitting packets across divergent processing paths–for example, sending packets through different downstream code paths as a result of routing lookups or IDS matches. This limits their use to simple linear pipelines, rather than the complex element graphs supported by Click. Snap draws inspiration from this work, and seeks to simply and efficiently integrate GPU processing into complex, fully-functional routers and other packet processors. Snap will be an enabler for future work in this area.

PTask [22] is a GPU framework that supports complex dataflow graphs; however, its model of data processing, which is based on Unix pipes, is different from the model used by a packet processor. In PTask, each processing element consumes its input data and produces one or more *new* streams of output data. Packet processing, in contrast, passes the *same* data, packets, through a series of elements that modify, annotate, drop, or deliver them. This leads to a different set of design decisions regarding how to allocate, store, manage, and reuse data memory.

Click has existing support for multiple processors [6]; however, this support was designed for the level of parallelism seen on PC CPUs; ie. dozens of cores, not hundreds or thousands. Route-Bricks [7] showed that Click-based software routers can be scaled up by using a network of PCs. In contrast, Snap focuses on the routing performance of a single PC, and is thus complementary to RouteBricks; by increasing each PC's throughput, Snap could be used to reduce the number of PCs needed in a RouteBricks-like routing cluster. Kim et al. demonstrated another strategy for improving Click performance by batching packets [13]. Here, batching refers to executing the *same element* on a set of packets in series, in contrast with standard Click, which executes a set of elements on the *same packet* in series. This work is also complementary to Snap, as it could be used to accelerate the CPU portions of a Snap pipeline, and could be driven by Snap's batch scheduling.

3. THE DESIGN AND IMPLEMENTATION OF SNAP

We designed Snap with two goals in mind: enabling *fast* packet processors through GPU offloading while preserving the *flexibility* in Click that allows users to construct complex pipelines from simple elements. Snap is designed to offload specific elements to the GPU: parts of the pipeline can continue to be handled by existing elements that run on the CPU, with only those elements that present computational bottlenecks re-implemented on the GPU. From the perspective of a developer or a user, Snap appears very similar to regular Click, with the addition of a new type of "batch" element that

can be implemented on the GPU and a set of adapter elements that move packets to and from batch elements. Internally, Snap makes several changes to Click in order to make this pipeline work well at high speeds. Several themes appear in our design choices. In many cases, we find that if we do "extra" work, such as making copies of packets in main memory, or passing along packets that we know will be discarded, we can decrease the need for synchronization and reduce our use of the relatively slow PCIe bus. We also find that scheduling parts of the pipeline asynchronously works well, and fits naturally with Click's native push/pull scheduling. In this section, we walk through the design and implementation of Snap, starting at a high level with the user-visible changes, and progressing through the lower level changes that stem from these high-level decisions.

3.1 Widening Click's Pipeline

As shown by the experiments in Section 2.3, in order to see large benefits from GPU offloading, we need to provide the GPU with relatively large batches of packets. In standard Click, the connection between elements is a single packet wide: the push() and pull() methods that pass packets between elements yield one packet each time they are invoked. To efficiently use a GPU in the pipeline, we added wider versions of the push() and pull() interfaces, bpush() and bpull(). These methods exchange a new structure called a *PacketBatch*, which will be described in more detail in the following section. We also made Click's Port class aware of these wider interfaces so that it can correctly pass *PacketBatch*es between elements. bpush() and bpull() belong to a new base class, BElement, which derives from Click's standard Element class.

In standard Click, to implement an element, the programmer creates a new class derived from Element and overloads the push() and pull() methods. This is still supported in Snap; in fact, most of our pipelines contain many unmodified elements from the standard Click distribution, which we refer to as "serial" elements. To implement a parallel element in Snap the programmer simply derives it from BElement and overrides the bpush() and bpull() methods.

A GPU-based parallel element is comprised of two parts: a GPU side, which consists of GPU kernel code, and a CPU side, which receives *PacketBatch*es from upstream elements and sends commands to the GPU to invoke the GPU kernel. Snap provides a GPURuntime object to help Click code interact with the GPU, which is programmed and controlled using NVIDIA's CUDA toolkit [18]. GPU-based elements interact with GPURuntime to request GPU resources such as memory. The GPU kernel is written in CUDA's variant of C or C++, and is wrapped in an external library that is linked with the element sources when compiling Snap. Typically, each packet is processed by its own thread on the GPU.

3.2 Batching Packets for the GPU

The BElement class leaves us with a design question: how should we collect packets to form *PacketBatch*es, and how should we manage copies of *PacketBatch*es between host and GPU memory? Our answer to this question takes its cue from the functioning of Click's Queue elements. Parts of a Click configuration operate in a push mode, with packets arriving from a source NIC; other parts of the configuration operate in pull mode, with packets being pulled along towards output NICs. At some point in the configuration, an adapter must be provided between these two modes of operation. The family of Queue elements plays this role. In practice, the way a packet is processed in Click is that it is pushed from the source NIC through a series of elements until it reaches a Queue, at which point it is deposited there and Click returns to the input NIC to process the next packet. On the output side of the Queue, the packet is dequeued and processed until it reaches the output NIC.

In an analogous manner, we have created a new element, Batcher, which collects packets one at a time from a sequential Element on one side and pushes them in batches to a BElement on the other side. A Debatcher element performs the inverse function. Implementing this functionality as a new element, rather than changing Click's infrastructure code, has three advantages. First, it minimizes the changes within Click itself. Second, it makes transitions between CPU and GPU code explicit; since there are overheads associated with packet batching, it is undesirable for it to be completely invisible to the user. Third, and most important, it means that batching and offloading are fully under the control of the creator of the Snap configuration—while we provide carefully-tuned implementations of Batcher and Debatcher elements, it is possible to provide alternate implementations designed for specific uses (such as non-GPU BElements) without modifying Snap.

Batcher has two important configuration parameters: *batch-size* and *timeout*:

- *batch-size* determines how many packets the Batcher collects before passing them to its output. A large batch exploits more parallelism, but can increase latency, since it takes longer to collect.
- *timeout* is used to keep a bound on latency; it determines the maximum amount of time Batcher waits before passing along a partial batch.

A batch of packets is represented by the *PacketBatch* structure, which is illustrated in Figure 3. A *PacketBatch* is associated not only with a collection of packets (represented by Click's Packet objects), but also with allocations of host and GPU device memory. Large consecutive buffers are used in host and GPU memory in order to enable efficient DMA transfers, minimizing the overhead of setting up transfers across the PCIe bus. The large buffers of a *PacketBatch* are split into small buffers, which contain the slices of packets (such as the headers) that are needed by the BElement(s). During batching, the Batcher is responsible for selectively copying data from Click's Packet object into the packet's host memory buffer, as described in Section 3.7.

3.3 Managing Host-Device Memory Copies

GPU code requires both input data and output results to reside in GPU memory, making it necessary to copy packets back and forth between main memory and the GPU across the PCIe bus. Snap factors this task out of the BElements that contain processing code: a HostToDeviceMemcpy element (provided as part of Snap) is placed between the Batcher and the first element that runs on the GPU. An analogous DeviceToHostMemcpy element
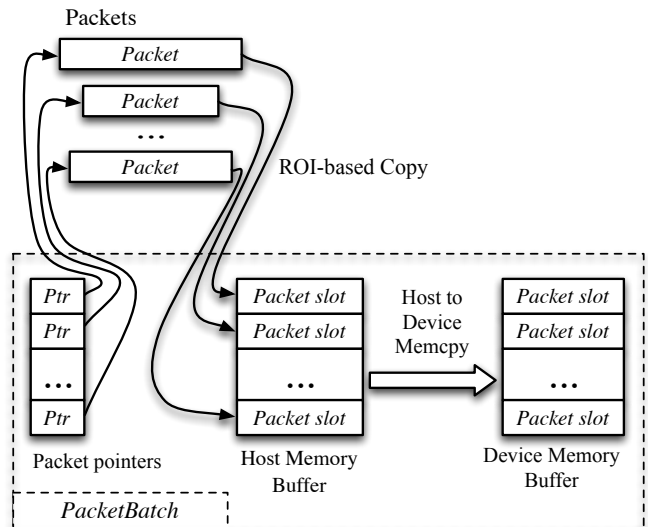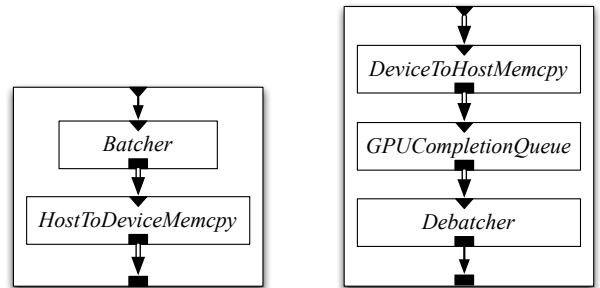

Figure 3: The *PacketBatch* structure. (See Section 3.7 for ROI)

(a) Batching packets for copying to the GPU

(b) In-order completion management and debatching

Figure 4: Batching and debatching elements. Serial interfaces are shown as simple arrows, wide interfaces as double arrows.

is placed before the Debatcher. Multiple GPU elements can be placed between a HostToDeviceMemcpy/DeviceToHostMemcpy pair, allowing the output ports of one to feed into the input ports of another without incurring a copy back to host memory. This design reduces the host-device packet copy times, reducing the overall memory copy overhead. These memory copy elements, along with the batching and debatching elements, can be seen in Figure 4.

3.4 Parallel Processing Without Reordering

Previous work on parallel packet processing often causes reordering among packets due to techniques such as load balancing and parallel dispatch across multiple cores [5, 7]. This can hurt TCP or streaming performance [16]. Snap, however, does not suffer from this problem: it waits for all threads in a GPU BElement to complete before passing the batch to the next element, so Snap does not reorder packets within a *PacketBatch*. Because asynchronous scheduling on the GPU (discussed in more detail in Section 3.8) may cause reordering of the *PacketBatch*es themselves, we add a GPUCompletetionQueue element between the DeviceToHostMemcpy and Debatcher elements. GPUCompletetionQueue keeps a FIFO queue of outstanding GPU operations, and does not release a *PacketBatch* downstream to the Debatcher until all previous *PacketBatch*es have been re-

leased, keeping them in order. Because `GPUCompletetionQueue` is simply an `Element` in the configuration graph, a configuration that is not concerned about reordering could simply provide an alternate element that releases batches as soon as they are ready.

3.5 Putting It Together: A GPU Pipeline

Figure 5 shows how all of these elements operate together. Packets enter one at a time from the top. They could come directly from the NIC or from sequential Click `Elements`. A `Batcher` collects them into a *PacketBatch* and sends them along to a `HostToDeviceMem-cpy` element, which initiates a copy to the GPU. This, and all other interaction with the GPU, is done through a `GPURuntime` element provided by Snap. The GPU element launches the GPU code kernel and yields to the `DeviceToHostMemcpy` element, which sets up a copy of the GPU element's results back to host memory; this will occur automatically when the GPU code completes execution. The `GPUCompletetionQueue` element waits for the preceding GPU operations to complete, and passes *PacketBatch*es, in FIFO order, to a `Debatcher`. The `Debatcher` then passes packets one by one into an output serial `Element`.

This picture tells only the part of the story that is visible at the level of a Snap configuration. The remainder of this section is devoted to a set of design decisions and optimizations that occur at lower levels to make these pieces work efficiently.

3.6 Handling Packet Divergence in Batches

Snap faces a problem not encountered by other GPU processing frameworks [9, 11, 25], namely the fact that packets in Click do not all follow the same path through the `Element` graph. `Elements` may have multiple output `Ports`, reflecting the fact that different packets get routed to different destination NICs, or that different sets of processing elements may be applied depending on decisions made by earlier elements. This means that packets that have been grouped together into a *PacketBatch* may be split up while on the GPU or after being returned to host memory. We encounter two main classes of packet divergence. In *routing or classification divergence*, the number of packets exiting on each port is relatively balanced; with *exception-path divergence* most packets remain on a "fast path" and relatively few are diverted for special processing. Packet divergence may also appear in two places: before the packets are sent to the GPU, or on the GPU, as a result of the decisions made by BElements.

Figure 6(a) shows an example of exception-path divergence before reaching the GPU: packets may be dropped after the TTL decrement if their TTLs reach zero. Figure 6(b) shows an example of routing divergence on the GPU. In this example, different IDS elements (likely applying different sets of rules) are used to process a packet depending on its next hop, as determined by IP routing lookup.

Divergence before reaching the GPU is one reason that we do not attempt to implement zero-copy in the *PacketBatch* structure. The effect of divergence early in the pipeline is memory fragmentation, giving us regions of memory in which only some packets need to be copied to the GPU. This problem is particularly pronounced in the case of routing/classification divergence. Copying all packets, even unnecessary ones, to the GPU would waste time and scarce PCIe bandwidth. Alternately, we could set up a number of small DMA transfers, covering only the necessary packets, but this results in high DMA setup overhead. Instead, we use the relatively plentiful memory bandwidth in host RAM to copy the necessary packets to one continuous region, which can be sent to the GPU with a single DMA transfer.

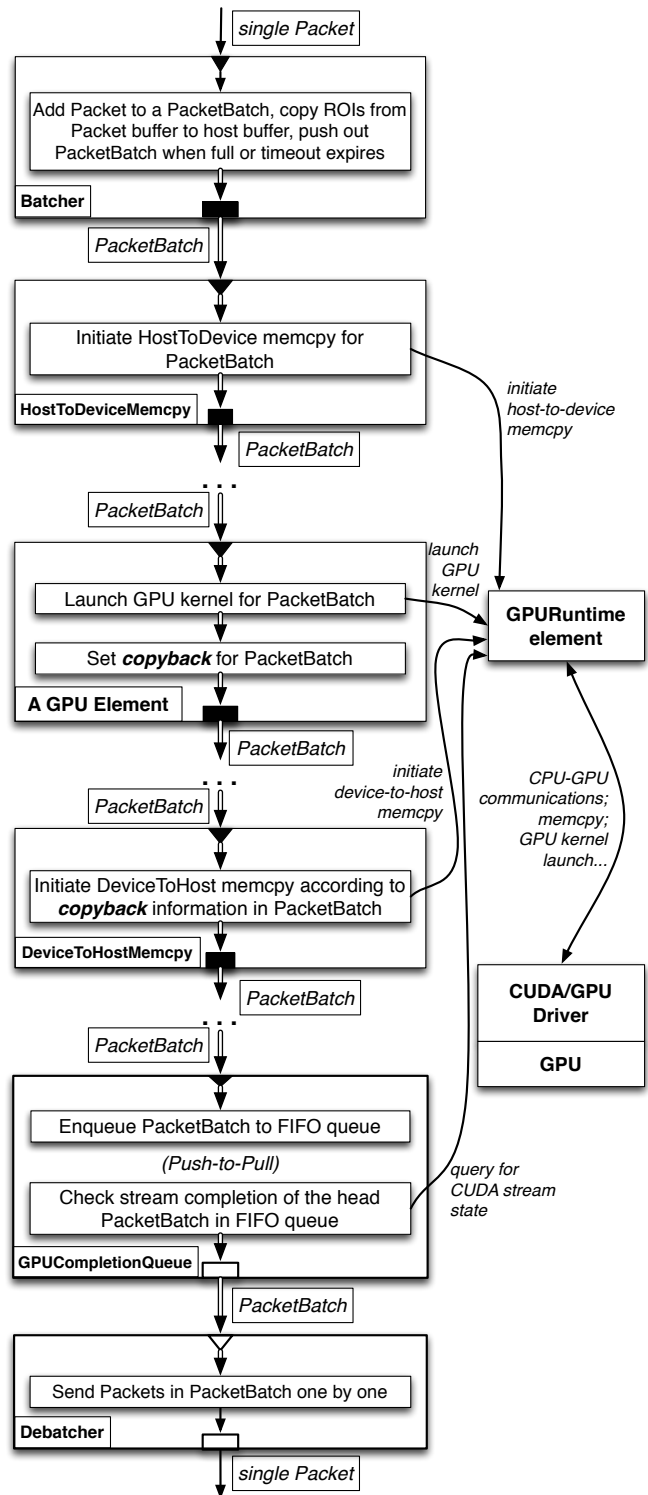For divergence that occurs on the GPU, our experiments show

Figure 5: A batch processing path with one GPU element.

that the overheads associated with splitting up batches and copying them into separate, smaller *PacketBatch*es are prohibitive, especially in the case of exception-path divergence. Assembling output *PacketBatch*es from selected input packets is also not concurrency-friendly: determining each packet's place in the output buffer requires knowledge of how many packets precede it, which in turn requires global
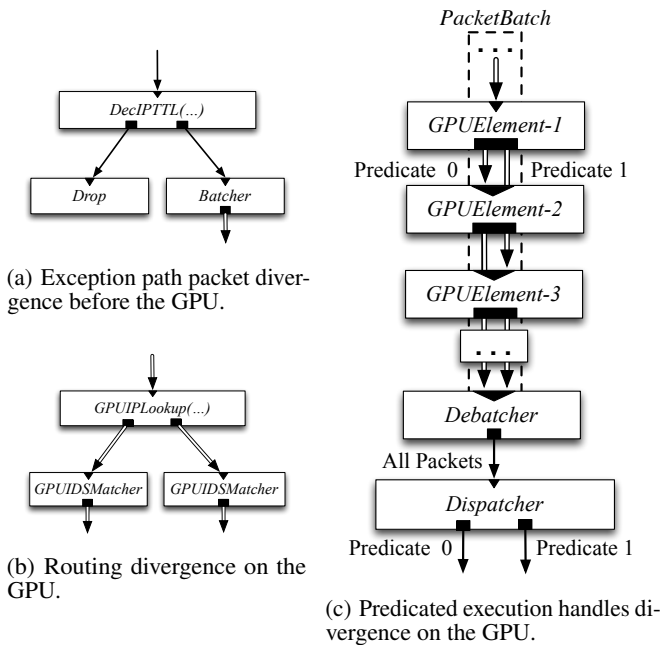
(a) Exception path packet divergence before the GPU.

(b) Routing divergence on the GPU.

(c) Predicated execution handles divergence on the GPU.

Figure 6: Handling divergent packet paths.

Figure 7: A Slicing example

3.7 Slicing for Efficient Offloading

Many packet processing algorithms, such as Ethernet switching, IP route lookup, packet classification, and IP header checksum computation, need only a portion of the entire packet. As a common case, many need only protocol headers. This means that when offloading, it is not always necessary to copy the entire packet to device memory. For example, IP route lookup needs only the destination IP address, so even for minimum-sized 64-byte packets, copying just the four-byte destination can save up to 94% of the memory copy bandwidth. This presents another opportunity to use relatively plentiful main memory bandwidth, by making a small copy, in order to save time and bandwidth on the more restrictive PCIe bus. To take advantage of this situation, we designed a slicing mechanism in *PacketBatch* to optimize the host-to-device packet copy performance.

Slicing (illustrated in Figure 7), allows GPU processing elements to specify the regions of packet data that they operate on, called Regions of Interest (ROIs). An ROI is a consecutive range in the packet data buffer, and a GPU processing element can have multiple ROIs spread throughout the packet. `Batcher` accepts ROI requests from its downstream `BElement`s and takes their union to determine which parts of the packet must be copied to the GPU. It allocates only enough host and device memory to hold these ROIs, and during batching, `Batcher` only copies data in these regions into a packet's host memory buffer. This reduces both the memory requirements for *PacketBatch*es and the overhead associated with copying them to and from the GPU. During debatching, ROIs are selectively copied back into the Click `Packet` structure. Slicing is another reason that we chose not use a zero-copy approach for *PacketBatch*es.

`Batcher` contains optimizations to avoid redundant copies in the case of ROIs that overlap and to combine `memcpy()` calls for consecutive ROIs to reduce function call overhead. For element developers' convenience, we have provided helper APIs for `BElement`s that allow the element to address packet data relative to its ROIs—the true offsets within the *PacketBatch* are computed transparently.

One problem associated with ROIs is that it may be difficult to describe the exact range in numeric values. For example, a `Classifier` element may need TCP port numbers, but the offset of this data within a packet is not constant due to the presence of IP option headers. To support this case, `Batcher` provides some special values to indicate variable offsets, such as the beginning of the TCP header or the end of the IP header. This also enables

serialization or synchronization. Therefore, Snap attaches a set of *predicate bits* to each packet in a *PacketBatch*—these bits are used to indicate which downstream `BElement`(s) should process the packet. Predicates are stored as annotations in Click's `Packet` structure. The thread processing each packet is responsible for checking and setting its own packet's predicate; this removes the need for coordination between threads in preparing the output. Because they are only used to mark divergence that occurs on the GPU, not before it, we save PCIe bus bandwidth by not copying predicate bits *to* GPU memory; we do, however, copy them *from* the GPU once a chain of `BElement`s is finished, since they are needed to determine the packets' next destination `Element`s on the CPU.

Figure 6(c) shows how this predicated execution works. The `GPUElement-1` element marks packets with either Predicate 0 or Predicate 1, depending on which downstream element should process them. The packets remain together in a single *PacketBatch* as they move through the element graph, but `GPUElement-2` only processes packets with Predicate 0 set and `GPUElement-3` only processes those with Predicate 1. Eventually, once they have left the GPU, the packets encounter a `Dispatcher` element, which sends them to different downstream destinations depending on their predicate bits. This arrangement can be extended to any number of predicate bits to build arbitrarily complicated paths on the GPU.

We have experimented with two strategies for using predicate bits: scanning all packets' predicates, and only launching GPU threads for the appropriate packets; and launching threads for all packets, and returning immediately from threads that find that their packet has the wrong predicate. We found it more efficient to launch threads for all packets: scanning packets for the correct predicates in order to count the number of threads adds to the startup overhead for the `BElement`. The savings in execution time that come from launching fewer threads are typically smaller than the overhead of scanning for the correct threads to launch, and it is faster to simply launch all threads. Because we run many threads per core, the threads that exit early do not necessarily waste a core.
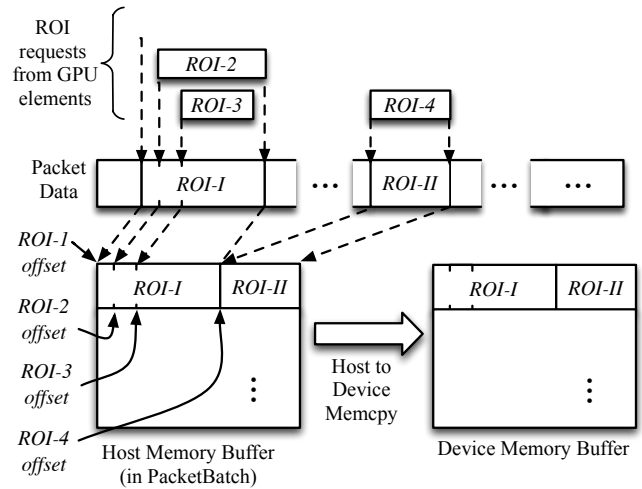
some special ROIs, such as ROIs that request the entire IP header including all IP options, or ROIs that cover the payload of the packet.

3.8 Asynchronous Scheduling of BElements

The host uses two main operations for controlling the GPU: initiating copies between host and device memory; and launching kernels on the GPU. Both can be done asynchronously, with the CPU receiving an interrupt when the copy or code execution completes. Multiple GPU operations can be in flight at once. Based on this, most GPU elements can work asynchronously on the CPU side: when a GPU element is scheduled by Snap, it issues appropriate commands to the GPU and schedules its downstream element by passing the *PacketBatch* immediately, without waiting for completion on the GPU. As a result, a push path or a pull path with GPU elements can keep pushing or pulling *PacketBatch*es without blocking on the GPU. This allows us to achieve very low latency at the beginning of the path, which is critical when packet rates are high—for example when receiving minimum-sized packets from a 10Gbps interface. The Click `FromDevice` element that receives packets from the NIC must disable interrupts while it pushes packets downstream to first `Queue` or `Batcher` element that they encounter; it is thus critical to have this path run as quickly as possible to avoid lost packets.

Snap uses CUDA "streams," which allow multiple memory copies and kernel executions to run concurrently. Each stream has a queue of operations which is run in FIFO order. Operations from different streams run concurrently and may complete in any order. We associate each *PacketBatch* with a unique stream. When the *PacketBatch* is first passed to a GPU element (typically, `HostToDeviceMemcpy`), it gets a stream assignment. Each subsequent BElement along the path asynchronously queues execution of its operation within the stream and passes control to the next BElement immediately, without waiting for the GPU. This sequence of events continues until control reaches a `GPUCompletetionQueue`, which is a *push-to-pull* element, much like a `Queue`. When a *PacketBatch* is pushed into the `GPUCompletetionQueue`, it simply adds the batch's stream to its FIFO queue and returns. When the `GPUCompletetionQueue`'s `bpull()` method is called, usually by a downstream `Debatcher`, it checks the status of the stream at the head of the FIFO. If the stream has finished, `bpull()` returns the *PacketBatch*; if not, it indicates to the caller that it has no packets ready.

3.9 Speeding Up Click's Packet I/O

Click includes existing support for integration with Netmap [20] for fast, zero-copy packet I/O from userspace. We found, however, that Click's design for this integration did not perform well enough to handle the packet rates enabled by Snap.

Netmap uses the multiqueue (RSS/MQ) support in recent NICs to enable efficient dispatch of packets to multiple threads or CPU cores. The queues maintained by Netmap in RAM are mapped to hardware queues maintained by the NIC; the NICs we use for our prototype fix each receive and transmit queue at 512 packets. When a packet arrives on the NIC, a free slot is found in a queue, and the NIC places the packet in a buffer pointed to by the queue slot. When the packet is passed to Click, the buffer, and thus the queue slot, remains unavailable until Click is either finishes with the packet (by transmitting it on another port or discarding it) or copies it out into another buffer. Since packets may take quite some time to be processed, they tie up these scarce queue slots, which can lead to drops. This problem is exacerbated in Snap, which needs to wait for suitably large batches of packets to arrive before sending them to the GPU. Click's solution is to copy packets out when it notices the Netmap queues getting full. It uses a single global memory pool for all threads, leading to concurrency problems. We found that

at the high packet rates supported by Snap, these copies occurred for nearly every packet, adding up to high overhead incurred for memory allocation and copying. Note that unlike our *PacketBatch* structure, which copies only regions of interest, the copies discussed here must copy the entire packet.

Unmodified Netmap gives the userspace application a number of packet buffers equal to the number of slots in the hardware queues; while kernel code can request more buffers, userspace code cannot. We added a simple system call that enables applications to request more packet buffers from Netmap. Though the size of the queues themselves remain fixed, Snap can now manipulate the queue slots to point to these additional buffers, allowing it to maintain a large number of in-process packets without resorting to copying. Snap maintains a pool of available packet buffers—when it receives a packet from the NIC, it changes the queue to point to a free packet buffer, and packet buffers are added back to the free pool when the packets they hold are transmitted or dropped. This eliminates packet buffer copying and overhead from complex memory allocation (`kmalloc()`), and we use per-thread packet buffer pools to avoid overhead from locking.

We also modified Click to pin packet I/O threads to specific cores. This is a well-known technique that improves cache behavior and interrupt routing when used with multiqueue NICs. Combined, these two optimizations give Snap the ability to handle up to 2.4 times as many packets per second as Click's I/O code—this improvement was critical for small packet sizes, where the unmodified packet I/O path was unable to pull enough packets from the NIC to keep the processing elements busy.

4. EVALUATION

We ran a set of experiments on Snap to evaluate how well it meets our original goals: can we write a useful set of elements that run on the GPU, can we compose them with each other and with CPU code, and can we achieve high forwarding rates?

Snap is forked from commit `9200a74` in the Click source repository [14]; "standard Click" experiments use this version. We used the Netmap release from August 13, 2012 and Linux kernel 3.2.16. Snap makes 2,179 lines of changes to Click itself, plus includes 4,636 lines of code for new elements and a 3,815 line library for interacting with the GPU. We modified only 180 lines of code in Netmap. The source for Snap, including our modifications to Netmap, can be downloaded from `https://github.com/wbsun/snap`

All experiments were performed on a PC with an Intel Core i7 930 quad-core CPU running at 2.8 GHz. It had 6 x 2 GB of DDR3-1600 triple channel memory having 38.4 GB/s of bandwidth, and an ASUS P6X58D-E motherboard with an Intel X58 chipset having 25.8 GB/s QPI in each direction for PCIe 2.0 devices. The GPU used was an NVIDIA TESLA C2070, which has 448 cores running at 1.15 GHz and 6 GB of GDDR5 RAM. The NICs are two Intel 82599EB dual-port 10 Gbps cards, for a total of four 10 Gbps ports. The GPU is connected via a 16-lane PCIe v2.0 slot and the NICs are each connected to 8-lane slots. All equipment used for evaluation is publicly available as part of the Emulab testbed [27]. Packets were generated at full line rate using a modified version of the packet generator that comes with the Netmap distribution, using a separate set of hosts in Emulab. Packet sizes reported are the Ethernet frame payload. When calculating throughput in Gbps, we add in the Ethernet preamble, SFD, CRC, and interframe gap, so that a reported rate of 10 Gbps represents 100% utilization on a 10 Gbps interface. Forwarding tables were designed such that all packets were forwarded back out the interface they arrived on. This ensured that all outgoing traffic was perfectly balanced so that any drops we observed were due to effects within the Snap host, rather

Configuration	Throughput	
Click 1 Path	4.55 Gbps	6.5 Mpps
Click 4 Paths (1 thread)	8.28 Gbps	11.8 Mpps
Click 4 Paths (4 threads)	13.02 Gbps	18.5 Mpps
Snap 1 Path	8.59 Gbps	12.2 Mpps
Snap 4 Paths	30.97 Gbps	44.0 Mpps

Table 2: Base Forwarding Performance of Snap and Click

than congestion on unbalanced outbound links.

4.1 Packet I/O Improvements

Our first set of experiments are simple microbenchmarks that evaluate the packet I/O optimizations described in Section 3.9. We measured the forwarding rate for minimum-sized (64 byte) packets using Click's Netmap packet I/O engine and Snap's improvements to that engine. These experiments use the simplest possible forwarder, which simply passes packets between interfaces with no additional processing. We test both a one-path arrangement, which passes packets from a single input NIC to a single output, and a four-path arrangement that uses all four NICs in our test machine. Click's existing Netmap support is not thread-safe, allowing only one packet I/O thread to be run. We added multithreading support to standard Click's Netmap code, and also report performance for four threads, one per NIC. Snap adds support for multiple threads per NIC, each using a different MQ/RSS queue, so we use sixteen threads for the Snap configuration.

The performance numbers are found in Table 2. Snap's improvements to the I/O engine introduce a 1.89x speedup for single path forwarding and 2.38x speedup for four-path forwarding. One interesting result is that Snap's four-path performance is not quite four times that of its single-path performance. This suggests that there may be room to improve the forwarding performance of Snap using more cores; our test CPU has four physical cores and hyperthreading, meaning that there are two I/O threads mapped to each hyperthreaded core.

4.2 Applications

We have implemented elements for three kinds of packet processing tasks. Each has a CPU and a GPU version:

GPUIPLookup: this GPU-based IP lookup element implements Click's standard IPRouteTable class using a radix tree. Its CPU counterpart is Click's RadixIPLookup element. For evaluation, we used a routing table dump from routeview. org [1] that has 167,000 entries.

SDNClassifier, GPUSDNClassifier: these two elements classify packets using seven fields from the Ethernet, IP, and UDP or TCP headers. Each entry assigns an action by forwarding the packet out of a specific outbound Port on the element. This is roughly analogous to the flow space matching used by many SDN forwarding schemes. SDNClassifier is the CPU version. The classification rule set is ClassBench [24]'s ACL1_10K filter set. We randomly assigned an action number to each rule.

IDSMatcher, GPUIDSMatcher: these two elements implement Aho-Corasick [2] string matching on packet payloads. The Aho-Corasick algorithm can match multiple patterns simultaneously by scanning the entire packet payload once. We used Snort's [21] rules for Mysql, Apache, Webapps, and PHP to simulate a set of rules for a Web application server.

We combine these elements to build three kinds of Snap configurations, each of which has both a GPU and a CPU version:

SDN Forwarder: This configuration includes only the SDNClassifier or its GPU counterpart. It simulates an SDN-like switch.

DPI Router: This configuration includes an IP lookup element (RadixIPLookup or GPUIPLookup) and a string matching element (IDSMatcher or GPUIDSMatcher) as the major processing elements. The intent is to simulate a router with a simple deep packet inspection firewall.

IDS Router: This configurations includes all three elements (IP lookup, IDS matcher, and SDN classifier) to simulate a more sophisticated router with complicated forwarding rules and intrusion detection.
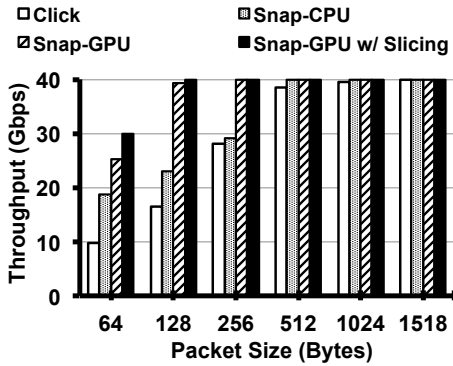
4.3 Application Performance

Using these applications, we compared the performance of four configurations: standard Click; Snap with only CPU Elements; Snap with GPU elements, but with packet slicing disabled; and Snap with all optimizations enabled. We experimented with a variety of packet sizes. Each experiment lasted at least one minute, and the numbers reported are the average of three runs. The results are shown in Figure 8.

The results show that Snap gets significant performance improvements over Click, particularly for small packet sizes. A significant fraction of this speedup comes from our I/O optimizations, which can be seen by comparing the bars for "Click" and "Snap-CPU:" 63% of the 3.1x speedup seen by the SDN forwarder on 64 byte packets comes from this source. Another jump comes from moving the processing-heavy elements to the GPU, with another modest increase with the addition of packet slicing. Snap is able to drive all four NICs at full line rate for all but the smallest packets: at and above 128 bytes, it gets full line rate for all configurations (with the exception of the IDS Router, which gets 39.6 GBps at 128 bytes). Snap is limited by the availability of PCIe slots in our test machine, which has 32 PCIe lanes: 16 are used by the GPU, and each dual-port NIC uses 8 lanes, meaning that we cannot add any more NICs. The results strongly suggest that Snap would be capable of higher bandwidth on a machine with more or faster PCIe lanes, but such hardware was not available for our tests. We thus leave exploration of Snap's full limits to future work and the availability of suitable hardware, but we are optimistic that it will be capable of exceeding 40 Gbps for large packets. Conversely, this result also means that there is headroom available to do more processing per packet than is performed by our example applications.

For minimum sized packets, Snap reaches 29.9 Gbps (75% of the full line rate) for the SDN forwarder; the primary cause of this limitation appears to be due to packet I/O, as the throughput seen in this experiment is very close to the trivial forwarder from Section 4.1. The other two GPU applications reach almost the same performance: 29.2 Gbps for the DPI router and 28.0 Gbps for the IDS Router. This matches our intuition, because with all of the complex processing done on the GPU, the CPU only needs to perform simple operations and can spend most of its time on packet I/O. When we use the CPU elements, both packet I/O and the processing algorithms need the CPU, and all three applications slow down significantly: the IDS Router gets 14.73 Gbps, a slowdown of 52% from the trivial forwarder. With the DPI and IDS Router configurations, standard Click is unable to reach much more than 20 Gbps, even for large packet sizes.
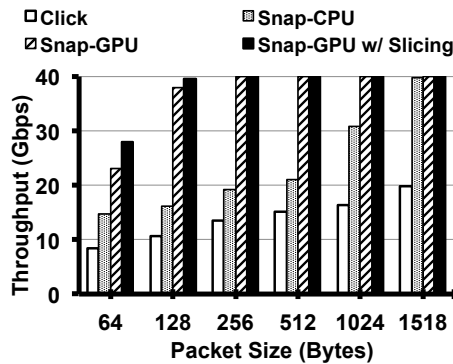
The ROI-based slicing mechanism makes a modest improvement in forwarding throughput. For example, the SDN forwarder sees a

(a) SDN Forwarder
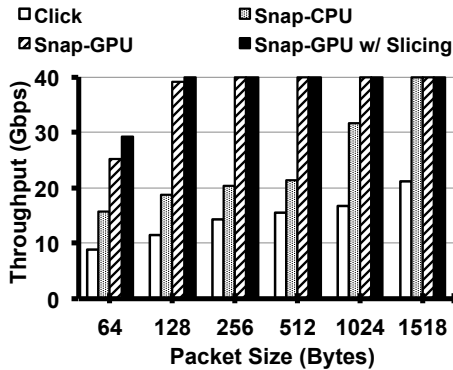
(b) DPI Router

(c) IDS Router

Figure 8: Performance of Click and Snap with three different applications.

13.7% increase in throughput for 64 byte packets. Slicing enables Snap to reach nearly the full rate supported by the packet I/O engine for small packets. At larger packet sizes, the improvement disappears because we have reached full line rate on the NICs.

4.4 Latency and Reordering

The most obvious drawback of batched processing is an increase in latency, since packets arriving at the beginning of a batch must wait for the batch to fill. To find out how much latency the batching mechanism adds to Snap, we measured round-trip time for 64-byte packets using both a CPU and GPU configuration. For the CPU-only configuration, we saw a mean latency of 57.5μs (min: 31.4μs,
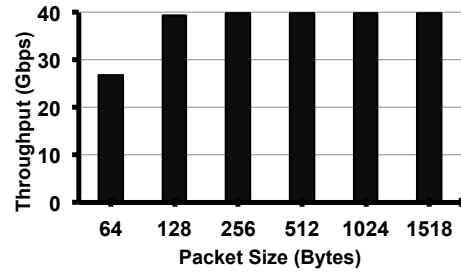

Figure 9: Forwarding performance when using a `GPUSDNClassifier` **that diverges to two** `GPUIDSMatcher` **elements.**

max: 320μs, σ: 25.7μs). For the GPU-based configuration with batched processing (*batch-size* 1024), the mean latency was 508μs (min: 292μs, max: 606μs, σ: 53.0μs); this represents an increase of less than one quarter of a millisecond in each direction. Reducing the *batch-size* from 1024 to 512, the latency reduces to 380.4μs on average, but throughput also drops from 28 Gbps to 24 Gbps. The additional latency added by batching for GPU elements is likely to be noticeable, but tolerable, for many LAN applications. On WAN links, this delay will be negligible compared to propagation delay.

As part of this experiment, we also checked for packet reordering. We define the RSS dispatching rules of our router NICs to send packets in the same UDP/TCP flow into the same NIC transmit queue, and in the Snap configuration, we connected each `FromDevice` element to a `ToDevice` with the same transmit and receive queue IDs. With these settings, we found no reordering in the packet stream.

4.5 Packet Divergence

To evaluate whether our design for handling divergent paths is effective, we built an IDS configuration that connects an `GPUSDNClassifier` element with two `GPUIDSMatcher` elements. The classifier marks each packet with a predicate indicating which of the two IDS elements is to process it; this simulates a scenario in which packets are to be handled by different IDSes depending on some property such as source, destination, or port number. In both this configuration and the IDS Router configuration from our earlier experiments, each packet is processed by one IDS element; the difference is that in the diverging configuration, there are two IDS elements, each of which processes half of the packets. Thus, we can expect that, if the overhead of our divergence handling strategy is low, the configuration with two `GPUIDSMatcher`s should achieve similar throughput to the configuration with a single one. We evaluated this diverging configuration with different packet sizes and measured the throughput, which is shown in Figure 9. The performance under divergence is very similar to the IDS Router result shown in Figure 8(c). It is only slightly slower at small packet sizes: the diverging configuration achieves 26.8 Gbps versus the IDS Router's 28.0 Gbps for 64 byte packets, 39.4 Gbps vs. 39.6 Gbps for 128 byte packets, and 39.9 Gbps vs. 40.0 Gbps for 256 bytes packets. At and above 512 byte packets, both achieve a full 40.0 Gbps. We conclude that the launch of extra GPU threads that have no work to do causes a slight slowdown, but the effects are minimal.

4.6 Flexibility and Modularity

Finally, we demonstrate that Snap can be used to build not only highly specialized forwarders, but also a complete standards compliant IP router. This task is simple, because such configurations already exist for Click. Specifically, we base our IP router off
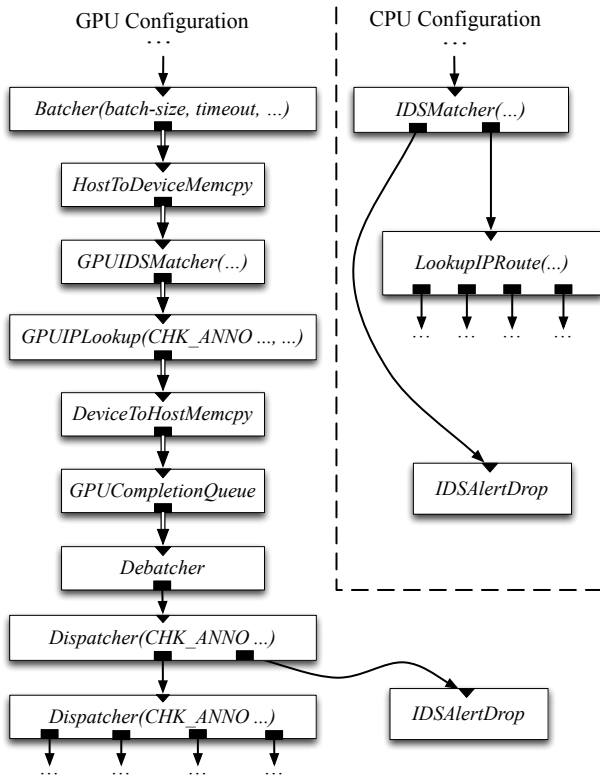
Figure 10: Major changes to the standard Click router to implement a GPU-based IDS router.

Figure 11: Fully functional IP router + IDS performance

we believe that the throughput can be significantly improved by moving some of these to the GPU and applying the techniques from Kim et al. [13] to optimize the remaining CPU portions of the configuration.

5. CONCLUSION AND FUTURE WORK

We have presented Snap, a packet processing system that builds upon Click to enable fast, flexible packet processing on GPUs. Snap expands Click's composable element structure, adding support for batch processing and offloading of computation. At small packet sizes (128 bytes), Snap increases the performance of a combined IP router, SDN forwarder, and IDS on commodity hardware from 10.6 Gbps to 39.6 Gbps. This performance increase comes primarily from two sources: an improved packet I/O engine for Click that takes advantage of multi-queue NICs, and moving computationally expensive processing tasks to the GPU. A trivial forwarder created with Snap can forward at a rate of 44.0 Mpps, while the complex SDN/IDS router reaches 90% of this rate (39.8 Mpps). These results suggest that there is likely potential for elements that are even more computationally complex than the ones we investigated, pointing to future work in complex packet processing. The fact that we are able to saturate all NICs in our test platform with such small packets suggests that it will be possible to reach even higher throughputs when PCIe 3.0 devices are available for testing, allowing us to double the number of NICs on a bus.

While some of the new `Element`s implemented for Snap, such as `HostToDeviceMemcpy` and `GPUCompletetionQueue`, are GPU-specific, the extensions we made to the Click architecture should be applicable to other parallel offload engines (such as network processors and programmable NICs) as well.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] RouteView Routing Table Dump. `http://www.read.cs.ucla.edu/click/routetabletest-167k.click.gz`.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an

of the configuration shown in Figure 8 of the Click paper [15], which includes support for error checking of headers, fragmentation, ICMP redirects, TTL expiration, and ARP. We replace the `LookupIPRoute` element with our `GPUIPLookup` element (and the accompanying `Batcher`, etc.), and add an IDS element to both the CPU and GPU configurations.

Due to the complexity of this router, we do not attempt to illustrate the entire Snap configuration here. Instead, we illustrate the major changes that we made to the standard Click router configuration in Figure 10. The left part of the figure shows our GPU processing path, and the right part is the original CPU route lookup path plus an `IDSMatcher` and its auxiliary alert element. This figure also shows a strategy for handling divergence on the GPU: the `GPUIDS-Matcher` sets predicates on packets depending on whether they should raise an alert, then pushes entire the *PacketBatch* downstream. The `GPUIPLookup` is assigned a *CHK_ANNO* argument, which is the predicate controlling processing of each packet. `GPUIPLookup` thus ignores packets flagged for alerts by the IDS, and divergence on the actual element graph is delayed until after the `Debatcher`, using a `Dispatcher` element.

The performance of the CPU-based and GPU-based full router configurations are shown in Figure 11. This fully-functional router with built-in IDS is able to achieve 2/3 of the performance of a trivial forwarder for minimum-sized packets, and almost full line rate (38.8 Gbps) for 512-byte and larger packets. This demonstrates the feasibility of composing complex graphs of CPU and GPU code, and shows that existing CPU Click elements can be easily used in Snap configurations without modification. The bottleneck in performance appears to be the large number of CPU elements in this configuration—there are 15 types of elements, some of which are duplicated sixteen times, once for each thread. As future work,
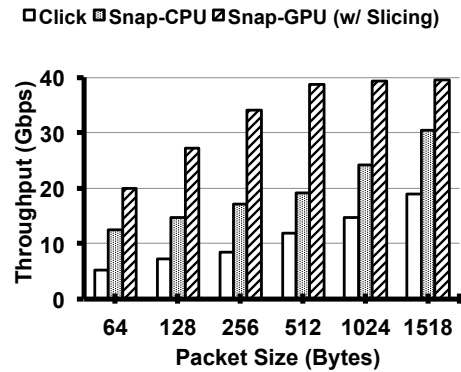
aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[3] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 219–230. ACM, 2008.

[4] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities. In *Proceedings of the HotNets Workshop*, 2003.

[5] J. C. R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Networking*, 7(6):789–798, Dec. 1999.

[6] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor PC router. In *Proceedings of USENIX ATC*, 2001.

[7] 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 *Proceedings of the 22nd SOSP*, 2009.

[8] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient support for evolvavle internetworking. In *Proceedings of NSDI*, 2012.

[9] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of ACM SIGCOMM*, 2010.

[10] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The architecture of a network level intrusion detection system. Technical report, Department of Computer Science, University of New Mexico, 1990.

[11] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A highly-scalable software-based intrusion detection system. In *Proceedings of ACM CCS*, 2012.

[12] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *Proceedings of NSDI*, 2011.

[13] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the Click modular router. In *Proceedings of the ACM Asia-Pacific Workshop on Systems*, 2012.

[14] E. Kohler. Click source code repository.

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

[16] M. Laor and L. Gendel. The effect of packet reordering in a backbone link on application throughput. *Network, IEEE*, 16(5):28 – 36, Sep/Oct 2002.

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

[18] NVIDIA Inc. CUDA C Programming Guide.

[19] NVIDIA Inc. Tesla GPUs. http://www.nvidia.com/object/tesla-servers.html.

[20] L. Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of USENIX ATC*, 2012.

[21] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX LISA*, 1999.

[22] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of SOSP*, 2011.

[23] W. Sun, R. Ricci, and M. L. Curry. GPUstore: harnessing GPU computing for storage systems in the OS kernel. In *Proceedings of SYSTOR*, 2012.

[24] D. E. Taylor and J. S. Turner. ClassBench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, June 2007.

[25] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. RAID 2008. Springer-Verlag.

[26] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire speed name lookup: A GPU-based approach. In *Proceedings of NSDI*, 2013.

[27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of OSDI*, 2002.

[28] Wikipedia. WAN optimization. http://en.wikipedia.org/wiki/WAN_optimization.