APUNet: Revitalizing GPU as Packet Processing Accelerator

Younghwan Go, Muhammad Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park

School of Electrical Engineering, KAIST

Abstract

Many research works have recently experimented with GPU to accelerate packet processing in network applications. Most works have shown that GPU brings a significant performance boost when it is compared to the CPU-only approach, thanks to its highly-parallel computation capacity and large memory bandwidth. However, a recent work argues that for many applications, the key enabler for high performance is the inherent feature of GPU that automatically hides memory access latency rather than its parallel computation power. It also claims that CPU can outperform or achieve a similar performance as GPU if its code is re-arranged to run concurrently with memory access, employing optimization techniques such as group prefetching and software pipelining.

In this paper, we revisit the claim of the work and see if it can be generalized to a large class of network applications. Our findings with eight popular algorithms widely used in network applications show that (a) there are many compute-bound algorithms that do benefit from the parallel computation capacity of GPU while CPU-based optimizations fail to help, and (b) the relative performance advantage of CPU over GPU in most applications is due to data transfer bottleneck in PCIe communication of discrete GPU rather than lack of capacity of GPU itself. To avoid the PCIe bottleneck, we suggest employing integrated GPU in recent APU platforms as a cost-effective packet processing accelerator. We address a number of practical issues in fully exploiting the capacity of APU and show that network applications based on APU achieve multi-10 Gbps performance for many compute/memory-intensive algorithms.

1 Introduction

Modern graphics processing units (GPUs) are widely used to accelerate many compute- and memory-intensive applications. With hundreds to thousands of processing cores and large memory bandwidth, GPU promises a great potential to improve the throughput of parallel applications. Fortunately, many network applications fall into this category as they nicely fit the execution model of GPU. In fact, a number of research works [29–31, 33, 34, 46, 47] have reported that GPU helps improve the performance of network applications.

More recently, however, the relative capacity of GPU over CPU in accelerating network applications has been questioned. A recent work claims that most benefits of GPU come from fast hardware thread switching that transparently hides memory access latency instead of its high computational power [32]. They have also shown that applying the optimization techniques of group prefetching and software pipelining to CPU code often makes it more resource-efficient than the GPU-accelerated version for many network applications.

In this paper, we re-examine the recent claim on the efficacy of GPU in accelerating network applications. Through careful experiments and reasoning, we make following observations. First, we find that the computational power of GPU does matter in improving the performance of many compute-intensive algorithms widely employed in network applications. We show that popular cryptographic algorithms in SSL and IPsec such as RSA, SHA-1/SHA-2, and ChaCha20 highly benefit from parallel computation cycles rather than transparent memory access hiding. They outperform optimized CPU code by a factor of 1.3 to 4.8. Second, while the CPU-based code optimization technique like G-Opt [32] does improve the performance of naïve CPU code, its acceleration power is limited when it is compared with that of GPU without data transfer. By the performance-per-cost metric, we find that GPU is 3.1x to 4.8x more cost-effective if GPU can avoid data transfer overhead. Third, we emphasize that the main bottleneck in GPU workload offloading lies in data transfer overhead incurred by the PCIe communication instead of lack of capacity in the GPU device itself. However, masking the DMA delay is challenging because individual packet processing typically does not require lots of computation or memory access. This makes it difficult for asynchronous data transfer (e.g., concurrent copy and execution of GPU kernel) to completely overlap with GPU kernel execution. Given that typical PCIe bandwidth is an order of magnitude smaller than the memory bandwidth of GPU, it is not surprising that data transfer on PCIe limits the performance benefit in GPU-accelerated network applications.

To maximize the benefit of GPU without data transfer overhead, we explore employing integrated GPU in Accelerated Processing Units (APUs) for network applications. APU is attractive in packet processing as it provides a single unified memory space for both CPU and GPU, eliminating the data transfer overhead in discrete GPU. In addition, its power consumption is only a fraction of that of typical CPU (e.g., 35W for AMD Carrizo) with a much smaller price tag (e.g., $100 to $150), making it a cost-effective accelerator for networked systems. The question lies in how we can exploit it as a high-performance packet processing engine.

However, achieving high performance in packet processing with APU faces a few practical challenges. First, in contrast to discrete GPU, integrated GPU loses the benefit of high-bandwidth GDDR memory as it has to share the DRAM with CPU. Since both CPU and GPU contend for the shared memory bus and controller, efficient use of memory bandwidth is critical for high performance. Second, the communication overhead between CPU and GPU to switch contexts and to synchronize data update takes up a large portion of GPU execution time. Unlike discrete GPU, this overhead is no longer masked by overlapped GPU kernel execution. Also, cache coherency in APU is explicitly enforced through expensive instructions since CPU and GPU employ a separate cache. This implies that the results by GPU threads are not readily visible to CPU threads without heavyweight synchronization operations or slow GPU kernel teardown.

We address these challenges in a system called APUNet, a high-performance APU-accelerated network packet processor. For efficient utilization of memory bandwidth, APUNet extensively exercises zero-copying in all stages of packet processing: packet reception, processing by CPU and GPU, and packet transmission. We find that the extensive zero-copying model helps extract more computational power from integrated GPU, outperforming the naïve version by a factor of 4.2 to 5.4 in IPsec. For low-latency communication between CPU and GPU, APUNet adopts persistent GPU kernel execution that keeps GPU threads running in parallel across a continuous input packet stream. Eliminating GPU kernel launch and teardown, combined with zero copying significantly reduces packet processing latency by 7.3x to 8.2x and waiting time to collect a batch of packets for parallel execution. In addition, APUNet performs "group synchronization" where a batch of GPU threads implicitly synchronize the memory region of the processed packets, which avoids heavy atomics instructions. It efficiently ensures coherent data access between CPU and GPU by checking for updated packet payload (e.g., non-zero values in HMAC digest) only when processing is completed, which improves the synchronization performance by a factor of 5.7.

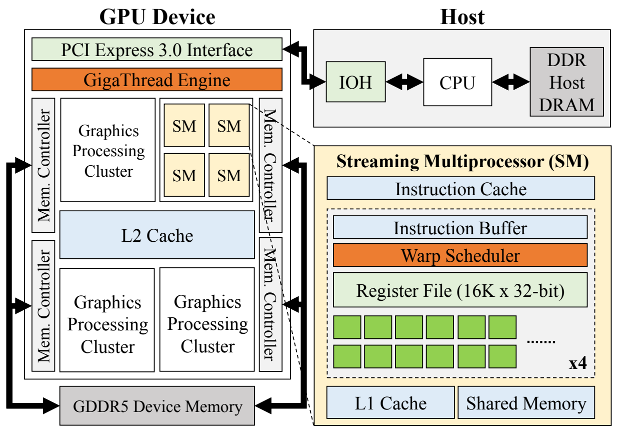


Figure 1: Typical architecture of discrete GPU

APUNet achieves both high-speed and cost-efficient network packet processing on several popular network applications. We find that many network applications, regardless of being memory- or compute-intensive, outperform CPU baseline as well as optimized implementations. APUNet improves G-Opt's performance by up to 1.6x in hashing-based IPv6 forwarding, 2.2x in IPsec gateway, 2.8x in SSL proxy, and up to 4.0x in NIDS adopting the Aho-Corasick pattern matching algorithm. We note that APUNet does not improve the performance of simple applications like IPv4 forwarding as the offloaded GPU kernel task is too small to overcome the CPU-GPU communication overhead.

2 Background

In this section, we describe the architecture of discrete and integrated GPUs and analyze their differences.

2.1 Discrete GPU

Most GPU-accelerated networked systems [29–31, 33, 34, 46, 47] have employed discrete GPU to enhance the performance of network applications. Typical discrete GPU takes the form of a PCIe peripheral device that communicates with the host side via PCIe lanes as shown in Figure 1. It consists of thousands of processing cores (e.g., 2048 in NVIDIA GTX980 [5]) and a separate GDDR memory with large memory bandwidth (e.g., 224 GB/s for GTX980). It adopts the single-instruction, multiple-thread (SIMT) execution model under which a group of threads (called a warp in NVIDIA or a wavefront in AMD hardware), concurrently executes the same instruction stream in a lock-step manner. Multiple warps (e.g., 4 in GTX980)

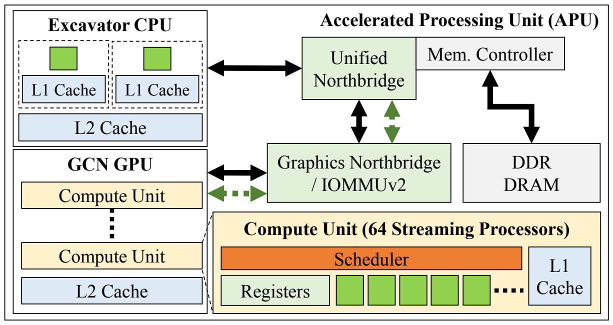


Figure 2: Architecture of integrated GPU (Carrizo APU)

are run on a streaming multiprocessor (SM) that shares caches (instruction, L1, shared memory) for fast instruction/data access and thousands of registers for fast context switching among threads. SMs are mainly optimized for instruction-level parallelism but do not implement sophisticated branch prediction or speculative execution as commonly available in modern CPU. High performance is achieved when all threads run the same basic block in parallel, but when a thread deviates from the instruction stream due to a branch, the execution of the other threads in a warp is completely serialized.

By the raw computation cycles and memory bandwidth, typical discrete GPU outperforms modern CPU. For example, NVIDIA GTX980 has 110x more computation cycles and 3.8x higher memory bandwidth than Intel Xeon E5-2650 v2 CPU. Such large computation power brings a clear performance advantage in massively-parallel workloads such as supercomputing [19, 38, 39] or training deep neural networks [21, 23, 24]. However, in other workloads that do not exhibit enough parallelism, GPU may perform poorly as it cannot fully exploit the hardware. Since our focus in this paper is packet processing in network applications, we ask a question: can GPU perform better than CPU in popular network applications? We attempt to answer this question in Section 3.

2.2 Integrated GPU

In recent years, integrated GPU has gained more popularity with extra computation power. Unlike discrete GPU, integrated GPU is manufactured into the same die as CPU and it shares the DRAM with CPU. Both Intel and AMD have a lineup of integrated GPU (e.g., AMD's Accelerated Processing Unit (APU) or Intel HD/Iris Graphics) that can be programmed in OpenCL. Figure 2 illustrates an example architecture of an integrated GPU inside AMD Carrizo APU [44].

Typical integrated GPU has lower computation capacity than discrete GPU as it has a smaller number of processing cores. For example, AMD Carrizo has 8 compute units (CUs) [1], each containing 64 streaming cores [2]. L1 cache and registers are shared among the streaming cores in a CU while L2 cache is used as a synchronization point across CUs. Despite smaller computation power, integrated GPU is still an attractive platform due to its much lower power consumption (e.g., 35W for AMD Carrizo) and a lower price tag (e.g., $100-$150 [2]). Later, we show that integrated GPU can be exploited as the most cost-effective accelerator by the performance-per-cost metric for many network applications.

The most notable aspect of integrated GPU comes from the fact that it shares the same memory subsystem with CPU. On the positive side, this allows efficient data sharing between CPU and GPU. In contrast, discrete GPU has to perform expensive DMA transactions over PCIe, which we identify as the main bottleneck for packet processing. For efficient memory sharing, OpenCL supports shared virtual memory (SVM) [9] that presents the same virtual address space for both CPU and GPU. SVM enables passing buffer pointers between CPU and GPU programs, which potentially eliminates memory copy to share the data. On the negative side, however, shared memory greatly increases the contention on the memory controller and reduces the memory bandwidth share per each processor. To alleviate the problem, integrated GPU employs a separate L1/L2 cache, but it poses another issue of cache coherency. Cache coherency across CPU and its integrated GPU can be explicitly enforced through atomics instructions of OpenCL, but explicit synchronization incurs a high overhead in practice. We address the two issues later in this paper. First, with extensive zero-copying from packet reception all the way up to GPU processing, we show that CPU and GPU share the memory bandwidth efficiently for packet processing. Second, we employ a technique that implicitly synchronizes the cache and memory access by integrated GPU to make the results of GPU processing available to CPU at a low cost.

3 CPU vs. GPU: Cost Efficiency Analysis

Earlier works have shown that GPU improves the performance of packet processing. This makes sense as network packet processing typically exhibit high parallelism. However, a recent work claims that optimized CPU code often outperforms the GPU version, with software-based memory access hiding. This poses a question: which of the two processors is more cost-effective in terms of packet processing? In this section, we attempt to answer this question by evaluating the performance for a number of popular network algorithms.

3.1 CPU-based Memory Access Hiding

Before performance evaluation, we briefly introduce the G-Opt work [32]. In that work, Kalia *et al.* have argued that the key performance contributor of GPU is not its high computation power, but its fast hardware context switch-

[1]CU is similar to NVIDIA GPU's SM.

[2]512 cores in total vs. 2048 cores in GTX980

ing that hides memory access latency. To simulate the behavior in CPU, they develop G-Opt, a compiler-assisted tool that semi-automatically re-orders the CPU code to execute memory access concurrently without blocking. G-Opt employs software pipelining to group prefetch the data from memory during which it context switches to run other code. They demonstrate that overlapped code execution brings a significant performance boost in IPv6 forward table lookup (2.1x), IPv4 table lookup (1.6x), multi-string pattern matching (2.4x), L2 switch (1.9x), and named data networking (1.5x). They also claim that G-Opt-based CPU code is more resource-efficient than the GPU code, and show that a system without GPU offloading outperforms the same system that employs it.

In the following sections, we run experiments to verify the claims of G-Opt. First, we see if popular network algorithms do not really benefit from parallel computation cycles. This is an important question since many people believe that earlier GPU-accelerated cryptographic works [29, 31] benefit from parallel computation of GPU. We see if G-Opt provides the same benefit to compute-intensive applications. Second, we see if G-Opt makes CPU more resource-efficient than GPU in terms of memory access hiding. Given that memory access hiding of GPU is implemented in hardware, it would be surprising if software-based CPU optimization produces better performance in the feature. Note that we do not intend to downplay the G-Opt work. Actually, we do agree that G-Opt is a very nice work that provides a great value in CPU code optimization. We simply ask ourselves if there is any chance to interpret the results in a different perspective.

3.2 Experiment Setup and Algorithms

We implement a number of popular packet processing tasks for the experiments. These include the algorithms evaluated in the G-Opt paper as well as a few cryptographic algorithms widely used in network security protocols and applications.

IP forwarding table lookup: IP forwarding table lookup represents a memory-intensive workload as typical IP forwarding table does not fit into CPU cache. IPv4 table lookup requires up to two memory accesses while IPv6 table lookup requires up to seven memory lookups with hashing. We use an IPv4 forwarding table with 283K entries as in PacketShader [29] and an IPv6 table with 200K randomized entries for experiments.

Multi-string pattern matching: Aho-Corasick (AC) multi-string pattern matching [18] is one of the most popular algorithms in network intrusion detection systems (NIDS) [15, 43] and application-level firewalls [7]. AC scans each payload byte and makes a transition in a DFA table to see if the payload contains one of the string patterns. It is a memory-intensive workload as each byte requires at least five memory accesses [22].

ChaCha20-Poly1305: ChaCha20-Poly1305 [25, 26] is a relatively new cipher stream actively being adapted by a number of Web browsers (e.g., Chrome) and Google websites. We select ChaCha20-Poly1305 as it is a part of AEAD standard for TLS 1.3, making AES-GCM and ChaCha20-Poly1305 the only options for future TLS [36]. ChaCha20 [26] is a stream cipher that expands 256-bit key into 2^{64} randomly accessible streams. It is mostly compute-bound as it does not require any table lookups. Poly1305 [25], a code authenticator, is also computation-heavy as it produces a 16-byte message tag by chopping the message into 128-bit integers, adding 2^{128} to each integer, then executing arithmetic operations on each integer.

SHA-1/SHA-2: SHA-1/SHA-2 are widely used for data integrity in network security protocols such as IPsec, TLS, PGP, SSH and S/MIME. SHA-1 is being rapidly replaced by SHA-2 as the attack probability on SHA-1 increases [14]. We select one of the standards of SHA-2, SHA-256, which produces a 256-bit digest by carrying out 64 rounds of compression with many arithmetic/bit-wise logical operations. Both SHA-1/SHA-2 are compute-intensive workloads.

RSA: RSA [13] is one of the most popular public key ciphers in TLS [48]. RSA is a compute-intensive workload as it requires a large number of modular exponentiations. We use 2048-bit RSA as 1024-bit key is deprecated [12].

Test platform: For experiments, we use recent CPU and GPU (both discrete and integrated GPU) as shown in Figure 3(a). For GPU code, we either follow the efficient implementations as described in earlier works [29–31, 47] or write them from scratch. We use the latest version of OpenSSL 1.0.1f for baseline CPU code, and apply G-Opt that we downloaded from a public repository. We feed a large stream of synthetic IP packets to the algorithms without doing packet I/O.

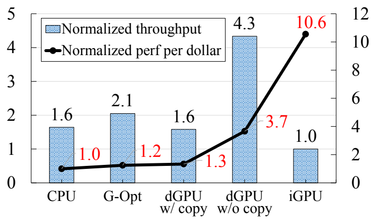
3.3 Performance Analysis

For each algorithm, we measure the throughputs of CPU baseline (CPU), G-Opt, discrete GPU with copy (dGPU w/ copy) and without copy (dGPU w/o copy), and integrated GPU (iGPU). "w/ copy" includes the PCIe data transfer overhead in discrete GPU while "w/o copy" reads the data from dGPU's own GDDR memory for processing. We include "w/o copy" numbers to figure out dGPU's inherent capacity by containing the PCIe communication overhead. Note, both "CPU" and "G-Opt" use all eight cores in the Xeon CPU while "dGPU" and "iGPU" use one CPU core for GPU kernel execution.

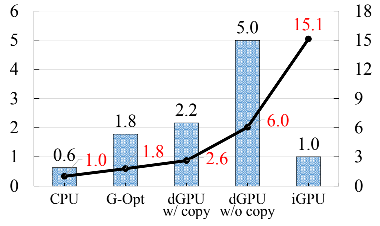
Performance-per-dollar metric: As each computing device has different hardware capacity with a varying price, it is challenging to compare the performance of heterogeneous devices. For fair comparison, we adopt the performance-per-dollar metric here. We draw the prices of hardware from Amazon [1] (as of September 2016)

CPU / Discrete GPU Platform	
CPU	Intel Xeon E5-2650 v2 (8 Core @ 2.6 GHz)
GPU	NVIDIA GTX980 (2048 Core @ 1.2 GHz)
RAM	64 GB (DIMM DDR3 @ 1333 MHz)
APU / Integrated GPU Platform	
CPU	AMD RX-421BD (4 Core @ 3.4 GHz)
GPU	AMD R7 Graphics (512 Core @ 800 MHz)
RAM	16 GB (DIMM DDR3 @ 2133 MHz)
OS	Ubuntu 14.04 LTS (Kernel 3.19.0-25-generic)

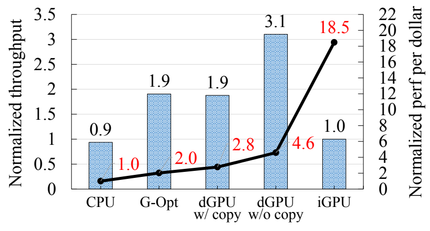
(a) CPU/GPU platforms for experiments



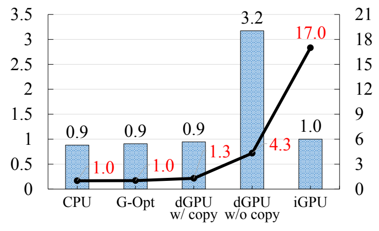
(b) IPv4 table lookup



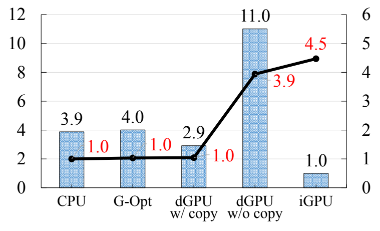
(c) IPv6 table lookup



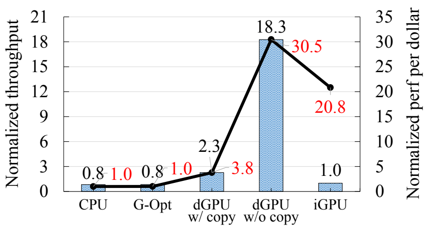
(d) Aho-Corasick pattern matching



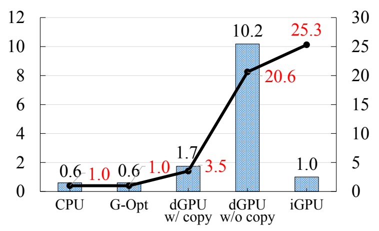
(e) ChaCha20



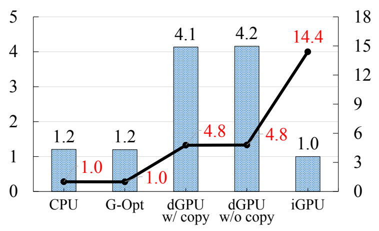
(f) Poly1305



(g) SHA-1



(h) SHA-256



(i) 2048-bit RSA decryption

Figure 3: Comparison of performance-per-dollar values (line) and per-processor throughputs (bar). Performance-per-dollar values are normalized by that of CPU while per-processor throughputs are normalized by that of iGPU.

Component	Price ($)	Details
CPU	1,143.9	8 CPU cores
Discrete GPU	840.0	GPU + 1 CPU core
Integrated GPU	67.5	GPU + 1 CPU core

Table 1: Price of each processor setup

and show them in Table 1. We estimate the price of iGPU as follows. We find that Athlon 860K CPU has an almost identical CPU specification as AMD Carrizo without integrated GPU. So, we obtain the price of iGPU by subtracting the price of Athlon 860K CPU from that of AMD Carrizo (e.g., \$50 = \$119.9 - \$69.99). For fairness, we include the price of one CPU core for GPU as it is required to run the GPU kernel. We note that the comparison by the performance-per-dollar metric should not be interpreted as evaluating the cost-effectiveness of CPU-based vs. GPU-based systems as it only compares the cost of each processor. However, we believe that it is still useful for gauging the cost-effectiveness of each processor for running network applications.

Performance comparison: Figure 3 compares the performance for each packet processing algorithm. Each line in a graph shows the performance-per-dollar values normalized by that of CPU. We use them as the main comparison metric. For reference, we also show per-device throughputs as a bar graph, normalized by that of iGPU.

We make following observations from Figure 3. First, G-Opt improves the performance of baseline CPU for memory-intensive operations (e.g., (b) to (d)) by a factor of 1.2 to 2.0. While these are slightly lower than reported by the original paper, we confirm that the optimization does take positive effect in these workloads. However, G-Opt has little impact on compute-intensive operations as shown in subfigures (e) to (i). This is because hiding memory access does not help much as computation capacity itself is the bottleneck in these workloads. Second, dGPU "w/o copy" shows the best throughputs by the raw performance. It outperforms G-Opt by a factor of 1.63 to 21.64 regardless of the workload characteristics. In contrast, the performance of dGPU "w/ copy" is mostly comparable to that of G-Opt except for compute-intensive operations. This implies that PCIe data transfer overhead is the key performance barrier for dGPU when packet contents have to be copied to dGPU memory. This is not surprising as the PCIe bandwidth of dGPU is more than an order of magnitude smaller than the memory bandwidth of dGPU. It also implies that dGPU itself has a large potential to

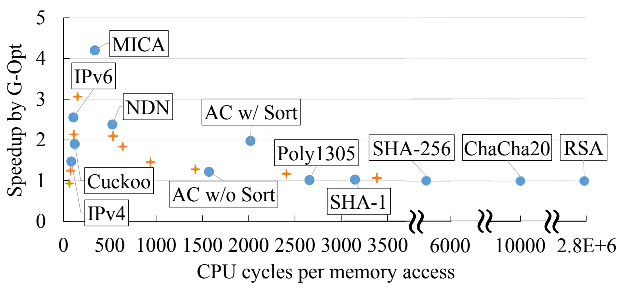


Figure 4: Correlation between speedup by G-Opt and the CPU-cycles-per-memory-access value. MICA [37] is an in-memory key-value store algorithm.

improve the performance of packet processing without the DMA overhead. Third, for compute-intensive operations, even dGPU "w/ copy" outperforms G-Opt by a large margin. By the performance-per-dollar metric, dGPU "w/-copy" brings 1.3x to 4.8x performance improvements over G-Opt for ChaCha20, SHA-1/SHA-2, and RSA. RSA shows the largest performance boost with dGPU as it transfers small data while it requires lots of computation. This confirms that compute-intensive algorithms do benefit from parallel computation cycles of GPU. Finally, and most importantly, we note that iGPU presents the best performance-per-dollar numbers in almost all cases. Even by the raw throughput, the performance of iGPU is often comparable to the baseline CPU with eight cores. This is because iGPU can fully export its computing capacity without PCIe overhead in dGPU.

Summary: We find that GPU-based packet processing not only exploits transparent memory access hiding, but also benefits from highly-parallel computation capacity. In contrast, CPU-based optimization helps accelerate memory-intensive operations, but its benefit is limited when the computation cycles are the key bottleneck. Also, dGPU has huge computation power to improve the performance of packet processing, but its improvement is curbed by the PCIe data transfer overhead. This turns our attention into iGPU as it produces high packet processing throughputs at low cost without the DMA bottleneck.

3.4 Analysis on CPU-based Optimization

We now consider how one can deterministically tell whether some packet processing task would benefit from CPU-based memory access hiding or not. Roughly speaking, if a task exhibits enough memory access that can be overlapped with useful computation, techniques like G-Opt would improve the performance. Otherwise, GPU offloading could be more beneficial if the workload can be easily parallelized.

To gain more insight, we measure the performance improvements by G-Opt for various packet processing algorithms, and correlate them with the trend of "CPU-cycles-per-memory-access", which counts the average number of CPU cycles consumed per each memory access. We measure the CPU-cycles-per-memory-access of each workload by counting the number of last-level-cache-misses of its baseline CPU code (without G-Opt optimization). For the measurement, we use Performance Application Programming Interface (PAPI) [11], a library that provides access to performance hardware counters.

Figure 4 shows the results of various algorithms. We use AMD CPU (in Figure 3(a)) for the measurement, but the similar trend is observed for Intel CPUs as well. For reference, we have a synthetic application that accesses memory at random such that we add arbitrary amount of computation between memory accesses. We mark this application as '+' in the graph. From the figure, we observe the following. First, we see performance improvement by G-Opt if x (CPU-cycles-per-memory-access) is between 50 and 1500. The best performance improvement is achieved when x is around 300 (e.g., MICA [37]). G-Opt is most beneficial if the required number of CPU cycles is slightly larger than that of memory access, reflecting a small amount of overhead to overlap computation and memory access. We think the improved cache access behavior after applying G-Opt [3] often improves the performance by more than two times. Second, we see little performance improvement if x is beyond 2500. This is not surprising since there is not enough memory access that can overlap with computation.

One algorithm that requires further analysis is Aho-Corasick pattern matching (AC w/ or w/o Sort). "AC w/o Sort" refers to the case where we apply G-Opt to the plain AC code, and we feed 1514B packets to it. "AC w/ Sort" applies extra optimizations suggested by the G-Opt paper [32] such that it pre-processes the packets with their packet header to figure out which DFA each packet falls into (called DFA number in [32]) and sorts the packets by their DFA number and packet length before performing AC. "AC w/o Sort" and "AC w/ Sort" show 1.2x and 2x performance improvements, respectively. While "AC w/o Sort" shows better performance improvement, it may be difficult to apply in practice since it needs to batch many packets (8192 packets in the experiment) and the performance improvement is subject to packet content.

4 APUNet Design and Implementation

In this section, we design and implement APUNet, an APU-accelerated network packet processing system. For high performance, APUNet exploits iGPU for parallel packet processing while it utilizes CPU for scalable packet I/O. We first describe the practical challenges in employing APU and then present the solutions that address them.

[3]For example, we see much better L1 cache hit ratio for MICA.

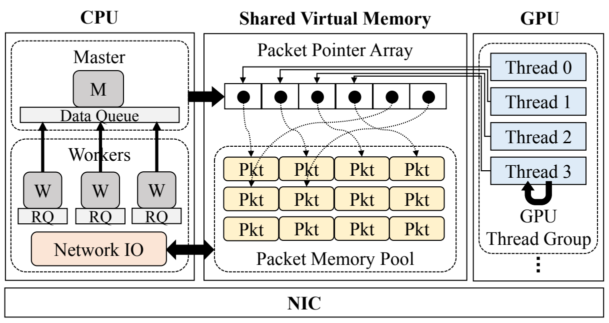


Figure 5: Overall architecture of APUNet. *M: master, W: worker, RQ: result queue.*

4.1 Challenge and Approach

APU enables efficient data sharing between CPU and GPU, but it also presents a number of practical challenges in achieving high performance. First, sharing of DRAM incurs contention on the memory controller and bus, which potentially reduces the effective memory bandwidth for each processor. Given that many packet processing tasks are memory-intensive and packet I/O itself consumes memory bandwidth, memory bandwidth is one of the key resources in network applications. Our approach is to minimize this shared memory contention by extensive zero-copy packet processing, which removes redundant data copy between CPU and GPU as well as NIC and application buffers. Second, we find that frequent GPU kernel launch and teardown incur a high overhead in packet processing. This is because GPU kernel launch and teardown in APU execute heavyweight synchronization instructions to initialize the context registers at start and to make the results visible to the CPU side at teardown. Our approach to address this problem is to eliminate the overhead with persistent thread execution. Persistent GPU thread execution launches the kernel only once, then continues processing packets in a loop without teardown. Lastly, APU does not ensure cache coherency across CPU and GPU and it requires expensive atomics operations to synchronize the data in the cache of the two processors. We address this challenge with group synchronization that implicitly writes back the cached data into shared memory only when data sharing is required. Our solution greatly reduces the cost of synchronization.

4.2 Overall System Architecture

Figure 5 shows the overall architecture of APUNet. APUNet adopts the single-master, multiple-worker framework as employed in [29]. The dedicated master exclusively communicates with GPU while the workers perform packet I/O and request packet processing with GPU via the master. The master and each worker are implemented as a thread that is affinitized to one of the CPU cores. Both CPU and GPU share a packet memory pool that stores an array of packet pointers and packet payload. They also share application-specific data structures such as an IP forwarding table and keys for cryptographic operations.

We explain how packets are processed in APUNet. APUNet reads incoming packets as a batch using the Intel DPDK packet I/O library [4]. Each worker processes a fraction of packets that are load-balanced by their flows with receive-side scaling (RSS). When a batch of packets arrive at a worker thread, the worker thread checks the validity of each packet, and enqueues the packet pointers to master's data queue. Then, the master informs GPU about availability of new packets and GPU performs packet processing with the packets. When packet processing completes, GPU synchronizes the results with the master thread. Then, the master notifies the worker thread of the results by the result queue of the worker, and the worker thread transmits the packets to the right network port. Details of data synchronization are found in the following sections.

4.3 Zero-copy Packet Processing

Zero-copy packet processing is highly desirable in APUNet for efficient utilization of the shared memory bandwidth. We apply zero-copying to all packet buffers and pass only the pointers between CPU and GPU for packet data access. In APU, one can implement zero-copying with SVM and its OpenCL API.

While SVM provides a unified virtual memory space between CPU and GPU, it requires using a separate memory allocator (e.g., `clSVMAlloc()`) instead of standard `malloc()`. However, this makes it difficult to extend zero copying to packet buffers as they are allocated in the DPDK library with a standard memory allocator. What is worse, the DPDK library allocates hugepages for packet buffers that map its virtual address space to a contiguous physical memory space. Unfortunately, DPDK does not allow turning off hugepage allocation as the rest of the code depends on the continuous physical memory space.

To address this problem, we update the DPDK code to turn off the hugepage mode, and make it use the SVM memory allocator to create packet buffers. We find that this does not result in any performance drop as address translation is not the main bottleneck in our applications. To avoid dealing with physical address mapping, we create an RDMA channel from one virtual address to another using the InfiniBand API [41] in the Mellanox NIC driver. We patch the DPDK code to support the driver in DPDK [40] and page-lock SVM-allocated buffers to be registered as the packet memory pool of the NIC. We then create an interface that exposes the SVM buffers to the application layer so that worker threads can access them to retrieve packets. While the current version depends on a particular NIC driver, we plan to extend the code to support other drivers. The total number of lines required for the modification is about 2,300 lines.

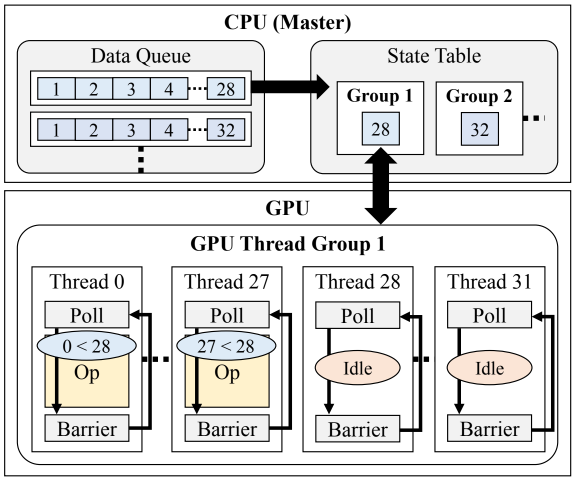


Figure 6: Persistent thread execution with a GPU thread group consisting of 32 threads

4.4 Persistent Thread Execution

GPU-based packet processing typically collects a large number of packets for high utilization of GPU threads. However, such a model comes with two drawbacks. First, it increases the packet processing delay to wait for the packets. Second, each GPU kernel launch and teardown incurs an overhead which further delays processing of the next batch of packets. The overhead is significant in APU as it has to synchronize the data between CPU and GPU at each GPU kernel launch and teardown.

To address the problem, we employ persistent GPU kernel execution. The basic idea is to keep a large number of GPU threads running to process a stream of input packets continuously. This approach has a great potential to reduce the packet processing delay as it completely avoids the overhead of kernel launch and teardown and processes packets as they are available. However, a naïve implementation could lead to serious underutilization of GPU as many GPU threads would end up executing a different path in the same instruction stream.

APUNet strives to find an appropriate balance between packet batching and parallel processing for efficient persistent thread execution. For this, we need to understand how APU hardware operates. Each CU in GPU has four SIMT units and each unit consists of 16 work items (or threads) that execute the same instruction stream in parallel [17]. So, we feed the packets to GPU so that all threads in an SIMT unit execute concurrently. At every dequeue of master's data queue, we group the packets in a multiple of the SIMT unit size, and pass them to GPU as a batch. We use 32 as the batch size in our implementation, and threads in two SIMT units (that we call as GPU thread group) process the passed packets in parallel. If the number of dequeued packets is smaller than the group size, we keep the remaining GPU threads in the group idle to make the number align with the group size. In this way, when the next batch of packets arrive, they are processed by a separate GPU thread group. We find that having idle GPU threads produces a better throughput as feeding new packets to a group of threads that are already active results in control path divergence and expensive execution serialization.

Figure 6 shows the overall architecture of persistent thread execution. CPU and GPU share a per-group *state* that identifies the mode of a GPU thread group as active (i.e., non-zero) or idle (i.e. zero). The value of the state indicates the number of packets to process. All GPU thread groups are initialized as idle and they continuously poll on the state until they are activated. When new packets arrive, the master thread finds an idle thread group and activates the group by setting the value of its group state to the number of packets to process. When the GPU thread group detects a non-zero value in its state, each GPU thread in the group checks whether it has a packet to process by comparing its local ID with the state value. Those threads whose local ID is smaller than the state value begin processing the packet inside the packet pointer array indexed by its unique global ID. Other threads stay idle waiting on a memory barrier. When all threads in the group finish execution and meet at the barrier, the thread with the minimum local ID switches the group state back to 0. The master in CPU periodically polls on the state, and when it sees that a group has completed, it retrieves the results and returns them to workers.

4.5 Group Synchronization

Another issue in persistent GPU threads lies in how to ensure cache coherency between CPU and GPU at a low cost. When GPU finishes packet processing, its result may not be readily visible in CPU as it stays in GPU's L2 cache, a synchronization point for GPU threads. In order to explicitly synchronize the update in GPU to main memory, OpenCL provides *atomics* instructions aligned with C++11 standard [16]. Atomics operations are executed through a coherent transaction path (dotted arrows in Figure 2) created by a coherent hub (CHUB) placed inside graphics northbridge [20]. Unfortunately, they are expensive operations as they require additional instructions such as locking, and CHUB can handle only one atomics operation at a time. As every atomics operation incurs an overhead, serial handling of requests from thousands of GPU threads would suffer from a significant overhead and degrade the overall performance.

We minimize the overhead by *group synchronization*. It attempts to implicitly synchronize the memory region of packets that a GPU thread group finished processing. For implicit synchronization, we exploit the LRU cache replacement policy of GPU [17] to evict dirty data items in GPU cache to main memory. We exploit idle GPU

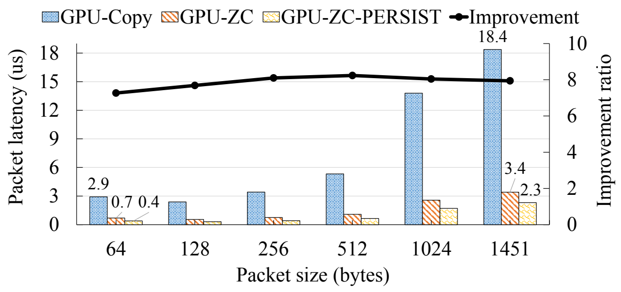


Figure 7: Packet latency with varying packet sizes

thread groups that are currently not processing packets such that they carry out dummy data I/O on a separate memory block and cause existing data in L2 cache to be evicted. While this incurs some memory contention, we find that synchronizing memory regions in a group is more efficient than synchronizing one at a time with atomics instructions. Note that idle threads (e.g., local ID larger than its group state value) in an active GPU thread group still wait on memory barrier since doing dummy data I/O will deviate them from instruction stream of active threads, making packet processing to be serialized. If all GPU thread groups are active, all GPU threads contend for GPU's cache and cause dirty data in cache to be flushed to main memory even without dummy data I/O. Finally, we guarantee data coherency between CPU and GPU by verifying the updates in packet payload (e.g., memory region for IPsec's digest contains non-zero values after processing).

4.6 Tuning and Optimizations

APUNet maximizes the performance of GPU kernel execution by applying a number of well-known optimizations, such as vectorized data accesses and arithmetic operations, minimization of thread group divergence, and loop unrolling. It also exploits GPU's high bandwidth local and private memory. We configure the number of persistent threads for maximum performance. We currently use a fixed dummy memory block size for group synchronization, and leave the design to dynamically configure the size depending on machine platform (e.g., different cache size) as a future work.

Finally, we enhance GPU's memory access speed by the hybrid usage of two SVM types: coarse- and fine-grained. Coarse-grained SVM provides faster memory access but needs explicit map and unmap operations to share the data between CPU and GPU, which incurs a high cost. Fine-grained SVM masks map/unmap overheads but shows longer memory access time due to pinned memory [27]. We thus allocate data structures that seldom change values (e.g., IP forwarding table) as coarse-grained SVM while we allocate frequently modified buffers (e.g., packet memory pool) as fine-grained SVM.

5 Evaluation

In this section, we evaluate APUNet to see how much performance improvement our design brings in packet processing. We then demonstrate the practicality of APUNet by implementing five real-world network applications and comparing the performance over the CPU-only approach.

5.1 Test Setup

We evaluate APUNet on the AMD Carrizo APU platform (shown as Integrated GPU Platform in Figure 3(a)) as a packet processing server. APUNet uses one CPU core as a master communicating with GPU while three worker cores handle packet I/O. We use a client machine equipped with an octa-core Intel Xeon E3-1285 v4 (3.50 GHz) and 32GB RAM. The client is used to either generate IP packets or HTTPS connection requests. Both server and client communicate through a dual-port 40 Gbps Mellanox ConnectX-4 NIC (MCX414A-B [3]). The maximum bandwidth of the NIC is 56 Gbps since it is using an 8-lane PCIev3 interface. We run Ubuntu 14.04 (kernel 3.19.0-25-generic) on the machines.

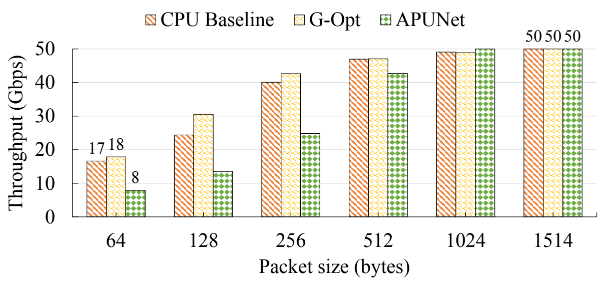
5.2 APUNet Microbenchmarks

We see if our zero-copy packet processing and persistent thread execution help lower packet processing latency, and gauge how much throughput improvement group synchronization achieves.

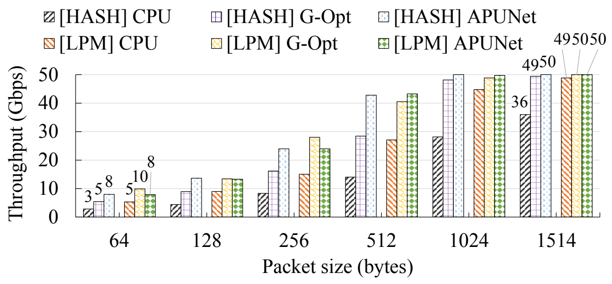
5.2.1 Packet Processing Latency

In this experiment, we measure the average per-packet processing latency from the time of packet arrival until its departure from the system. We evaluate the following three configurations: (i) *GPU-Copy*: a strawman GPU version with data copying between CPU and GPU address spaces (standard memory allocation in CPU with hugepages), (ii) *GPU-ZC*: GPU with zero-copy packet processing, and (iii) *GPU-ZC-PERSIST*: GPU with zero-copy packet processing *and* persistent thread execution including group synchronization. We use IPsec as packet processing task for the experiments, and use 128-bit AES in CBC mode and HMAC-SHA1 for the crypto suite.

Figure 7 compares the packet processing latency for each configuration with varying packet sizes. We set the largest packet size to 1451B, not to exceed the maximum Ethernet frame size after appending an ESP header and an HMAC-SHA1 digest. As expected, GPU-Copy shows the highest packet processing latency, which suffers from data transfer overhead between CPU and GPU. On the other hand, we see that GPU-ZC reduces the packet latency by a factor of 4.2 to 5.4 by sharing packet buffer pointers. GPU-ZC-PERSIST further reduces the latency by 1.8 times, thanks to elimination of per-batch kernel launch and teardown overheads. In total, APUNet successfully reduces the packet latency by 7.3 to 8.2 times, and shows an impressive reduction in packet processing time.



(a) IPv4 forwarding



(b) IPv6 forwarding

Figure 8: Performance of IPv4/IPv6 table lookup and packet forwarding

Moreover, it outperforms the CPU-only model by more than 2x in throughput in real-world network evaluation (Figure 9(a)).

5.2.2 Synchronization Throughput

We next evaluate the performance improvement of group synchronization when compared with synchronization with atomics operations. For this, we perform a stress test in which we pass 64B IP packets to GPU that is running an IPsec kernel code. We choose an IPsec module as our default GPU kernel since it completely updates the IP payload during encryption. Hence the entire packet content needs to be synchronized back to the CPU side. Note that zero-copy packet processing and persistent thread execution are implemented as baseline in both cases. Through measurement, we observe that APUNet achieves 5.7x throughput improvement with group synchronization (0.93 Gbps vs. 5.31 Gbps) as atomically synchronizing all dirty data serially via CHUB results in a significant performance drop. We find that APUNet significantly reduces the synchronization overhead between CPU and GPU and maintains high performance with a small cost of introducing additional memory usage from dummy data I/O.

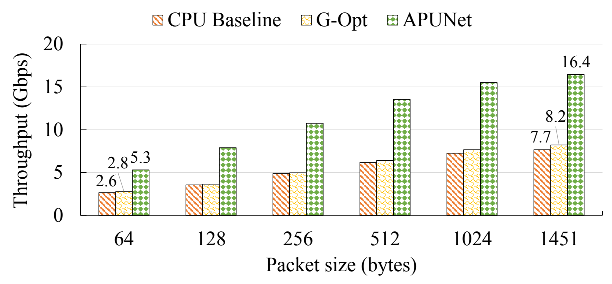
5.3 APUNet Macrobenchmarks

We determine the practicality of APUNet in real-world network applications by comparing the throughputs of CPU baseline and G-Opt enhanced implementations. We begin by briefly describing the setup of each application and then discuss the evaluation results.

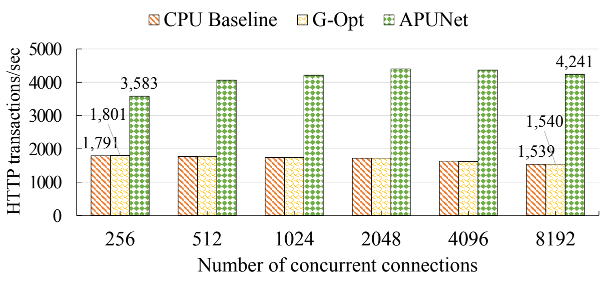
IPv4/IPv6 packet forwarding: We have a client machine transmit IPv4/IPv6 packets with randomized destination IP addresses, and APUNet forwards the packets after carrying out IPv4/IPv6 table lookups. We use the same IPv4/IPv6 forwarding tables as we used for experiments in Section 3. For IPv6 table lookup, we evaluate a hashing-based lookup algorithm as well as longest prefix matching (LPM)-based algorithm. Figure 8 shows the forwarding throughputs with varying packet sizes. We see that G-Opt effectively hides memory accesses of IPv4/IPv6 table lookups, improving CPU baseline performance by 1.25x and 2.0x, respectively. Unfortunately, we find that APUNet performs worse in IPv4 forwarding due to a number of reasons. First, as IPv4 table lookup is a lightweight operation, the communication overhead between CPU and GPU takes up a relatively large portion of total packet processing time. Second, APUNet dedicates one master CPU core for communicating with GPU and employs only three CPU cores to perform packet I/O, which is insufficient to match the IPv4 forwarding performance of four CPU cores. Lastly, there is a performance penalty from the contention for a shared packet memory pool between CPU and GPU. In contrast, IPv6 packet forwarding requires more complex operations, APUNet outperforms hashing-based CPU implementations by up to 1.6*x*. We note that the LPM-based CPU implementations are resource-efficient as it does not require hashing, and its G-Opt performance is mostly comparable to that of APUNet.

IPsec gateway and SSL proxy: We now show how well APUNet performs compute-intensive applications by developing an IPsec gateway and an SSL reverse proxy. Our IPsec gateway encrypts packet payload with the 128-bit AES scheme in CBC mode and creates an authentication digest with HMAC-SHA1. Both CPU baseline and its G-Opt version of the gateway exploit Intel's AES-NI [6], which is an x86 instruction set that runs each round of AES encryption/decryption with a single AESENC/AESDEC instruction. We implement the GPU version of the IPsec gateway in APUNet by merging AES-CBC and HMAC-SHA1 code into a single unified kernel for persistent execution. Figure 9(a) shows the evaluation results. As discussed in Section 3, G-Opt fails to improve the CPU baseline performance as AES is already optimized with hardware instructions and HMAC-SHA1 operation is compute-bound. On the other hand, APUNet delivers up to 2.2*x* performance speedup, achieving 16.4 Gbps with 1451B packets.

For SSL reverse proxying, we evaluate how many SSL handshakes APUNet can perform by offloading compute-intensive RSA operations to GPU. The client machine generates encrypted SSL requests using Apache HTTP



(a) IPsec gateway



(b) SSL reverse proxy

Figure 9: Performance of compute-intensive applications. For IPsec, we use 128-bit AES in CBC mode with HMAC-SHA1. For SSL proxying, we use 2048-bit RSA, 128-bit AES in CBC mode, and HMAC-SHA1 as a ciphersuite.

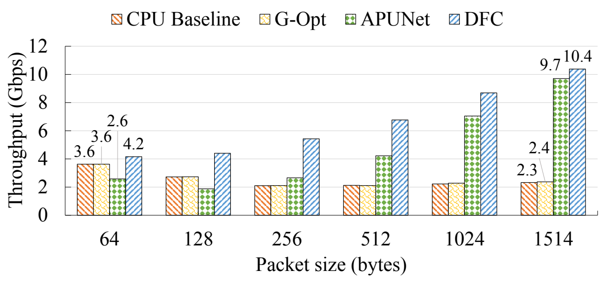


Figure 10: Performance of network intrusion detection system.

benchmark tool (ab) v2.3, that are then sent to APUNet. APUNet runs both SSL reverse proxy and Web server (nginx v1.4.6 [8]) where the proxy resides in the middle to translate between encrypted and plaintext messages. For SSL, we use TLS v1.0 with a 2048-bit RSA key, 128-bit AES in CBC mode, and HMAC-SHA1 as ciphersuite. Figure 9(b) shows the number of HTTP transactions processed per second with an increasing number of concurrent connections. Like IPsec, CPU-based optimization does not help much while our GPU-based implementation provides a performance boost for compute-intensive RSA and SHA-1 operations. Note that we have not applied persistent kernels for SSL as SSL requires switching across multiple crypto operations. Nevertheless, APUNet outperforms CPU-based approaches by 2.0x-2.8x.

Network intrusion detection system: We port a signature-based network intrusion detection system (NIDS) to APUNet. Our APUNet-based NIDS uses the Aho-Corasick (AC) pattern matching algorithm to scan ingress network traffic for known attack signatures. For evaluation, we port a full NIDS [30] that uses a Snort-based [43] attack ruleset to monitor malicious activity in a network and apply G-Opt to the AC pattern matching code for the G-Opt version. Under heavy stress, it is capable of offloading pattern matching tasks to GPU.

Figure 10 shows the throughputs against innocent synthetic traffic with randomized payload and varying packet sizes. Interestingly, we find that the full NIDS no longer benefits from G-Opt, different from 1.2x performance improvement of AC pattern matching ("AC w/o Sort") in Figure 4. This is because a fully functional NIDS would access a large number of data structures (e.g., packet acquisition, decoding and detection modules) during packet analysis and can cause data cache contention, resulting in eviction of already cached data by other prefetches before they are used. One may try to implement loop interchanging and sorting by DFA ID and packet length optimizations on all data structures as in [32] (described in Section 3.4), but this is still challenging to apply in practice as it requires batching a large number of packets (8192). In contrast, APUNet requires batching of a smaller number of packets in multiples of an SIMT unit size (e.g., 32). As a result, APUNet helps NIDS outperform AC-based CPU-only approaches by up to 4.0x for large packets. For small-sized packets, the computation load becomes small (only 10B of payload is scanned for 64B packets), making GPU offloading inefficient similar to IPv4 forwarding.

One notable thing is that DFC [22]-based NIDS outperforms APUNet -based NIDS by 7% to 61%. DFC is a CPU-based multi-string pattern matching algorithm that uses cache-friendly data structures and significantly reduces memory access at payload scanning [22]. Given that the synthetic traffic is innocent, almost all byte lookups would be hits in CPU cache as DFC uses a small table for quick inspection of innocent content. This shows that a resource-efficient algorithm often drastically improves the performance without the help of GPU. We leave our GPU-based DFC implementation as our future work.

6 Related Works

We briefly discuss previous works that accelerate network applications by optimizing CPU code or offloading the workload to GPU.

CPU code optimization: RouteBricks [28] scales the software router performance by fully exploiting the parallelism in CPU. It parallelizes packet processing with multiple CPU cores in a single server as well as distributing the workload across a cluster of servers, and achieves 35 Gbps packet forwarding performance with four servers. DoubleClick [33] improves the performance of the Click router [35] by employing batching to packet I/O and computation. Using PSIO [10], DoubleClick batches the received packets and passes them to a user-level Click element to be processed in a group, showing 10x improvement to reach 28 Gbps for IPv4 forwarding. Similarly, we show that our APUNet prototype with recent CPU achieves up to 50 Gbps performance for IPv4/IPv6 packet forwarding. Kalia *et al.* [32] have introduced the G-Opt framework that applies memory access latency hiding to CPU code. It implements software pipelining by storing the context information in an array, then executing prefetch-and-switch to another context whenever memory access is required. Although beneficial for memory-intensive workloads, our cost-efficiency analysis shows that G-Opt is limited in improving the performance of compute-intensive applications .

GPU offloading: Most GPU-based systems have focused on exploiting discrete GPU for accelerating packet processing. PacketShader [29] is the seminal work that demonstrates a multi-10 Gbps software router by offloading workload to discrete GPU, showing close to 40 Gbps for 64B IPv4/IPv6 forwarding, and 20 Gbps for IPsec. MIDeA [47] and Kargus [30] implement GPU-based, high-performance NIDSes (10-33 Gbps). MIDeA parallelizes packet processing via multiple NIC queues and offloads pattern matching tasks to GPU whereas Kargus can offload both pattern matching and regular expression matching workloads to GPU. SSLShader [31] optimizes SSL crypto operations by exploiting GPU parallelization and demonstrates 3.7x-6.0x SSL handshake performance improvement over a CPU-only approach.

There have also been attempts to build a general GPU-based network framework. GASPP [46] integrates common network operations (e.g., flow tracking and TCP reassembly) into GPU to consolidate multiple network applications. NBA [34] extends the Click router with batched GPU processing and applies adaptive load balancing to dynamically reach near-optimal throughputs in varying environments. Although a great platform for achieving high performance, we find that discrete GPU is less cost-efficient than integrated GPU for many network applications as they suffer heavily from data transfer overhead, having RSA as a notable exception.

More recently, Tseng *et al.* [45] developed a packet processing platform using Intel GT3e integrated GPU. They employ a continuous thread design similar to our persistent thread execution but uses a shared control object supported by Intel Processor Graphic Gen8 to synchronize the data between CPU and GPU. As a result, they show a small performance improvement of 2-2.5x over 1 CPU core for IPv4/IPv6 based applications. PIPSEA [42] is another work that implements an IPsec gateway on top of APU. PIPSEA designs a packet scheduling technique that sorts the packets by their required crypto suite and packet lengths to minimize control-flow divergence. In contrast to our work, PIPSEA does not address architectural overheads of APU such as memory contention and communication overhead, showing 10.4 Gbps throughput at 1024B packet size, which is lower than 15.5 Gbps of APUNet on a similar hardware platform. To the best of our knowledge, APUNet delivers the highest performance for a number of network applications on a practical integrated GPU-based packet processor.

7 Conclusion

In this work, we have re-evaluated the effectiveness of discrete and integrated GPU in packet processing over CPU-based optimization. With the cost efficiency analysis, we have confirmed that CPU-based memory access hiding effectively helps memory-intensive network applications but it has limited impact on compute-bound algorithms that have little memory access. Moreover, we observe that relative performance advantage of CPU-based optimization is mostly due to the high data transfer overhead of discrete GPU rather than lack of its inherent capacity. To avoid the PCIe communication overhead, we have designed and implemented APUNet, an APU-based packet processing platform that allows efficient sharing of packet data. APUNet addresses practical challenges in building a high-speed software router with zero-copy packet processing, efficient persistent thread execution, and group synchronization. We have demonstrated the practicality of APUNet by showing multi-10 Gbps throughputs with low latency for a number of popular network applications. We believe that APUNet will serve as a high performance, cost-effective platform for real-world applications.

Acknowledgments

We would like to thank our shepherd Fabián Bustamante and anonymous reviewers of NSDI'17 for their insightful comments and suggestions on the paper. We also thank Anuj Kalia at CMU for in-depth discussion on experiment results and further suggestions, and Seong Hwan Kim and Dipak Bhattacharyya at AMD, Inc., for their help in understanding the internals of APU and OpenCL. This work was supported in part, by the ICT Research and Development Program of MSIP/IITP, Korea, under Grant B0101-16-1368, [Development of an NFV-inspired networked switch and an operating system for multi-middlebox services], Grant R0190-16-2012, [High Performance Big Data Analytics Platform Performance Acceleration Technologies Development], and Grant R-20160222-002755, [Cloud-based Security Intelligence Technology Development for Customized Security Service Provisioning].

References

[1] Amazon. https://www.amazon.com.

[2] AMD A10-7860K-AD786KYBJCSBX. https://www.amazon.com/AMD-Processors-Radeon-Graphics-A10-7860K-AD786KYBJCSBX/dp/B01BF377W4.

[3] ConnectX-4 Lx EN Cards. http://www.mellanox.com/page/products_dyn?product_family=219&mtag=connectx_4_lx_en_card.

[4] DPDK: Data Plane Development Kit. http://dpdk.org/.

[5] GeForce GTX 980 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications.

[6] Intel Advanced Encryption Standard Instruction (AES-NI). https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni.

[7] ModSecurity: Open Source Web Application Firewall. https://www.modsecurity.org.

[8] NGINX. https://www.nginx.com.

[9] OpenCL 2.0 Shared Virtual Memory Overview. https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-overview.

[10] Packet I/O Engine. http://shader.kaist.edu/packetshader/io_engine/.

[11] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[12] Phasing out Certificates with 1024-bit RSA Keys. https://blog.mozilla.org/security/2014/09/08/phasing-out-certificates-with-1024-bit-rsa-keys/.

[13] Public-Key Cryptography Standards (PKCS) 1: RSA Cryptography Specifications Version 2.1. https://tools.ietf.org/html/rfc3447.

[14] Researchers Warn Against Continuing Use Of SHA-1 Crypto Standard. http://www.darkreading.com/vulnerabilities---threats/researchers-warn-against-continuing-use-of-sha-1-crypto-standard/d/d-id/1322565.

[15] Suricata Open Source IDS/IPS/NSM engine. http://suricata-ids.org/.

[16] Using OpenCL 2.0 Atomics. https://software.intel.com/en-us/articles/using-opencl-20-atomics.

[17] White Paper. AMD Graphics Core Next (GCN) Architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[18] Aho, Alfred V. and Corasick, Margaret J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.

[19] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[20] D. Bouvier and B. Sander. Applying AMD's "Kaveri" APU for Heterogeneous Computing. https://www.hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-11-day1-epub/HC26.11-2-Mobile-Processors-epub/HC26.11.220-Bouvier-Kaveri-AMD-Final.pdf.

[21] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine learning Library for Heterogeneous Distributed Systems. In *Proceedings of the NIPS Workshop on Machine Learning Systems (LearningSys)*, 2016.

[22] B. Choi, J. Chae, A. Jamshed, K. Park, and D. Han. DFC: Accelerating Pattern Matching for Network Applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[23] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *Proceedings of the NIPS Workshop BigLearn*, 2011.

[24] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[25] D. J. Bernstein. A state-of-the-art message-authentication code. https://cr.yp.to/mac.html.

[26] D. J. Bernstein. The ChaCha family of stream ciphers. https://cr.yp.to/chacha.html.

[27] B. Dipak. Question about Fine grain and Coarse grain in OpenCL 2.0. https://community.amd.com/thread/169688.

[28] 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 ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[29] S. Han, K. Jang, K. Park, and S. B. Moon. PacketShader: a GPU-accelerated Software Router. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[30] M. 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 the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[31] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[32] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[33] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The Power of Batching in the Click Modular Router. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*, 2012.

[34] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[35] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, August 2000.

[36] V. Krasnov. It takes two to ChaCha (Poly). https://blog.cloudflare.com/it-takes-two-to-chacha-poly/.

[37] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[38] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[39] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[40] O. Matz. Patchwork [dpdk-dev,v3,22/35] mempool: support no hugepage mode. http://www.dpdk.org/dev/patchwork/patch/12854/.

[41] Mellanox Technologies. Introduction to InfiniBand. http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[42] J. Park, W. Jung, G. Jo, I. Lee, and J. Lee. PIPSEA: A Practical IPsec Gateway on Embedded APUs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[43] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 1999.

[44] SAPPHIRE Technology. AMD Merlin Falcon DB-FP4. http://www.sapphiretech.com/productdetial.asp?pid=F3F349C0-5835-407E-92C3-C19DEAB6B708&lang=eng.

[45] J. Tseng, R. Wang, J. Tsai, S. Edupuganti, A. W. Min, S. Woo, S. Junkins, and T.-Y. C. Tai. Exploiting Integrated GPUs for Network Packet Processing Workloads. In *Proceedings of the IEEE Conference on Network Softwarization (NetSoft)*, 2016.

[46] G. Vasiliadis and L. Koromilas. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[47] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[48] W3Techs. Usage of SSL certificate authorities for websites. https://w3techs.com/technologies/overview/ssl_certificate/all.