International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

GPU Accelerated Network Packet Forwarding

Sujay Kanungo

Independent Researcher
Boston, USA
sujay.kanungo@gmail.com

Abstract:
The integration of Graphics Processing Units (GPUs) into network packet forwarding architectures has emerged as a transformative approach to enhancing performance and efficiency in data processing. This paper presents a comprehensive overview of recent advancements in utilizing GPUs for packet forwarding tasks, focusing on architectural innovations, performance metrics, and broader implications for network systems. By leveraging the parallel processing capabilities of GPUs, researchers have developed innovative solutions that address challenges such as high-speed data handling, low-latency processing, and effective resource management. The findings highlight the potential of GPU-based architectures to significantly improve packet forwarding rates and reduce latency, ultimately leading to more robust network infrastructures. This study also discusses the implications of these advancements for future network designs, emphasizing the importance of hybrid systems that combine traditional processing methods with modern GPU technology.

Keywords: Networks, Packet Forwarding, GPU.

I. INTRODUCTION

Interest in high-performance packet forwarding is growing due to rising demand for broadband access and increased interest in high-speed networking in computationally-intensive fields such as cloud computing, grid computing, and multimedia distribution. This has prompted widespread research into alternative architectures for high-performance switches and routers, with significant research investment being applied to networks and services. Particle forwarding is a basic, important process in networking. In switches and routers, all packets passing through must be forwarded [1].

Packet forwarding refers to moving a packet from one port to another. A packet sent from one host reaches a switch first, and the switch forwards the packet appropriately. In general, packet forwarding is classified into routing and switching. Routing concerns wide area networks (WANs) and packet forwarding based on network layer addresses. For example, routing packets from one country to another is routing [2]. Switching is a local area network (LAN) concept based on data link layer addresses. Multiple devices may exist on the same LAN, and the data link layer address, may, therefore, be used by switches to filter packet forwarding appropriate to the destination. Routers and switches performing such tasks are network devices, yet their working characteristics are quite different, requiring different SOC (System on Chip) designs for each.

Graphics processing unit-based packet forwarding and classification have become a focal point of networking research. In packet forwarding, incoming packets traverse a series of operations to determine output ports or modify header fields. Researchers have extensively analysed packet-forwarding schemes in computing architectures tailored for high-level parallelism and energy efficiency. Graphics processing units serve as dedicated computation devices that exploit parallelism, perform multi-data operations, and ensure high throughput with low energy consumption, making them attractive for packet processing.

Packet-forwarding workloads, comprising parsing and lookup phases, exhibit parallelism and data-level independence, rendering them suitable for graphics processing.

II. FUNDAMENTALS OF PACKET FORWARDING

Packet forwarding is a fundamental process in computer networking that enables data to travel from a source device to its intended destination across interconnected networks. At its core, packet forwarding is the mechanism by which routers and switches examine incoming data packets and decide the most efficient path for them to follow. This process ensures seamless communication across local networks, the internet, and large-scale enterprise infrastructures.

Each data packet contains two essential components: a header and a payload. The header includes critical information such as source and destination IP addresses, protocol type, and other control fields that help networking devices make forwarding decisions. When a packet arrives at a router, the router reads the destination IP address and consults its routing table—a database of known network paths. Based on this table, the router determines the next hop, or the immediate device to which the packet should be sent, moving it one step closer to its final destination.

Packet forwarding typically operates on two primary models: destination-based forwarding and policy-based forwarding. In destination-based forwarding, decisions rely solely on the destination IP address. Policy-based forwarding, however, considers additional factors such as application type, source address, or quality-of-service requirements, allowing for more dynamic and specialized routing.

Modern networking equipment employs various forwarding techniques, including packet switching, fast switching, and Cisco Express Forwarding (CEF)—a high-performance, hardware-optimized method widely used in enterprise networks. These techniques aim to reduce latency, improve throughput, and maintain scalability, especially in environments with heavy or unpredictable traffic loads.

Overall, packet forwarding plays an essential role in ensuring reliable and efficient data communication. Understanding its principles helps network engineers design robust infrastructures capable of supporting diverse applications, ranging from everyday web browsing to real-time services such as voice, video, and cloud computing.

III. GPU ARCHITECTURE AND ITS RELEVANCE IN PACKET PROCESSING

Driven by the exponential growth of mobile Internet traffic, the amount of data transmitted through core switches continues to rise. Network operators face increasingly complex demands while migrating toward software-defined networking. However, the packet processing and forwarding capacity of core network devices, particularly within enterprise and data-center scenarios, remains a major bottleneck. The forwarding capacity of routers varies by topology and application. Therefore, it is necessary to investigate forwarding architectures that achieve high throughput, low power consumption, flexibility, and cost-effective scaling.

Graphics processing units (GPUs) have emerged as a relevant hardware alternative for forwarding data packets. By leveraging abundant floating-point and integer processing performance, high data transfer rates, and flexible programming models, tools such as Cuda and OpenCL, GPUs are being increasingly incorporated into high-throughput packet processing. Differences between routers and graphics cards, such as single-cycle port queuing and on-chip port memory within routing schematics, actually justify the use of GPU facilities for certain operations.

In graphics processing, three types of parallelism coexist: Single Instruction, Multiple Data (SIMD), Single Instruction, Multiple Threads (SIMT), and data-parallelism. Each of these different types may

apply to one-dimensional data streams. Each of these types may also apply to one-dimensional data streams, such as point, line, and surface projections and modeling, Texture mapping, shading, etc. [3]. Concurrently, routers must accomplish numerous packet handling tasks, including packet parsing, flow classification, header checking, and statistics gathering. Each task has the potential to deliver large-throughput streaming and mass-parallel capabilities. Graphics frameworks are able to configure millions of parallel data flow to execute the per-packet cycle, thus providing adequate conditions for the implementation of GPU computation.

A. Parallelism Models in GPUs

With the rapid advancement of high-speed networks, packet forwarding has become one of the most important processing techniques in networking systems. Forwarding consists of inspecting the packet headers of incoming packets and deciding on the next output port based on a forwarding table stored in memory. Various semiconductor technologies have been developed to accelerate forwarding processing in routers and OpenFlow switches. In conjunction with these efforts, GPUs have gained increasing interest as cost-effective accelerators for high-performance networking tasks. Due to their relatively low price, widespread availability, and massive parallel computing architecture, GPUs have shown great potential for efficiently accelerating forwarding processing in the routing path of networking systems. The core forwarding processing of packets can be extracted and offloaded to a GPU while using a CPU to handle other control tasks. An attractive feature is the straightforward implementation of these mechanisms using the existing parallel processing capability of GPUs, making it an appealing option for designs requiring high performance.

B. Memory Hierarchy and Data Locality

Efficient data movement is paramount for high-performance packet processing on a GPU. Packet-processing kernels typically require bandwidths exceeding 1 TB/s when processing 64-byte packets on contemporary, single-GPU setups. Such impressive bandwidths pose serious hurdles to packet-duplicate detection and information gathering associated with TCP, stateful processing, flow classification, and similar operations. For real-time packet processing, the GPU cannot afford a single packet to be processed until all packets in flight have traversed that core. The highly parallel structure of GPUs affords the capability to keep many packet-processing elements operating concurrently despite the memory bottleneck.

Parallelism improves effective bandwidth by ensuring that, even when several accesses stall each kernel, active kernels remain on the chip, therefore keeping such engines supplied with processed packets. Kernel instructions can be reordered in the system by employing multi-pass architectures to circumvent additional dependency stalls. Such architectures also permit the communication of information about packets processed in prior passes. At the same time, as kernels proceed with one pass, other packets in flight can be dropped or marked for duplicate removal while additional entries are allocated. The limited storage capacity available for packet and state information reduces the overall duplication yet enables additional operations to be performed. Owing to the high compute–bandwidth ratio found in packet workloads, emulating stencil patterns can reduce pressure on the memory system, thus enhancing overall throughput [4].

GPU memory access patterns during unclamped packet processing typically adhere to a periodic, irregular data-flow model. Another longstanding approach calls for processing rich internal token data structures that represent parsed packets. Prior work has highlighted that programming frameworks supporting continuous streams naturally align with the periodic nature of packet data flows, enabling widespread installation in the packet-processing ecosystem. The short byte-wise format of network packets allows relatively few compiled data types to facilitate straightforward adaptation across hardware or network formats. High repeat rates for individual byte patterns also support effective lossless data compression, permitting an entire packet to be retained or a much-abridged replica [5].

C. Programming Models and Toolchains

Graphics Processing Units (GPUs) accelerate high-throughput packet processing through established programming paradigms with parallel task execution and specialized high-level Application Programming Interfaces (APIs) developed to effectively harness architecture-specific features. These models allow writing high-level code without detailed architecture knowledge while enabling compiler optimizations. Packet processing functions are specified in OpenCL kernels running on some NVIDIA GPUs with degree of precision similar to PacketShader for a variety of operations.

In addition to OpenCL, OpenGL also permits high-throughput packet processing and leverages more widely supported graphics hardware. For packet processing tasks beyond shader capabilities, graphics hardware can enhance overall performance through dedicated co-processors. Further tools like CUDA and MatCUDA provide corresponding acceleration for packet processing already available as public software.

IV. MAPPING PACKET FORWARDING TASKS TO GPU

Packet Parsing

Packet parsing is crucial for determining the layout of a packet and extracting the correct bit fields from its header [6]. Packet parsers vary widely in complexity due to the diversity of protocols, leading to inefficient off-the-shelf solutions. However, packet parsing translates naturally to the SIMD model expected by GPUs due to its data-parallel, memory-access–efficient structure. A single packet bitstream can thus be composed of a stream of packets identified by start, end, and offset indicators.

Header Inspection

Header inspection consists of reading the header, extracting the field values, and updating statistics related to the header. Multiple header fields can be processed simultaneously, since different fields of a packet are independent from a data-dependency standpoint. Dense headers also emerge fairly often, such as in Ethernet, IPv4, and TCP packets. Typically, extracted header fields are grouped into a fixed-size data structure that contains the most useful fields and towards which stateless classifiers are designed.

Flow Classification

Flow classification determines the Action associated with a given incoming packet based on its header field values. To install the Action for a flow, the header field values that characterize the flow need to be stored. The Action associated with a packet can be bound to its header field values to construct classifiers that are both stateless and can be evaluated in GPU multiprocessing kernels.

Steering Decision

Steering decision, or steering policy, is the choice of which output port to send a packet, which may be derived from the Action chosen. An incoming packet then finds an Action and the corresponding output port among many, so that the process is independent and just requires a Hash.

Stateful Processing

Stateful processing implies that a certain Action associated with a flow may change over time. In a stateful system, a flow may go through a series of temporal states. States are typically stored inside a solution object whose reference is set together with the Action variable. The processing of an Action then involves the examination of the state indicator, the value of the Action variable, and might carry out modifications to the state object. A natural candidate for modelling such a high-level description of stateful processing is a State machine.

A. Packet Parsing and Header Inspection

Packet parsing and header inspection involve analysing packet headers to extract routing, protocol, and addressing information for effective network processing. Filtering and classification tasks follow header inspection. Efficient packet capture architectures, including parallel hardware solutions, enhance performance in high-speed networking environments [7].

Certain field values, such as type-of-service (ToS) and class of service (CoS), are utilized to describe packet treatment and govern forwarding behavior. Forwarding state, including source and destination

addresses, port assignments, Virtual Local Area Network (VLAN) information, Policing classification, Quality of Service (QoS), and packet drops, remain fixed for short intervals. Subsequently, a constant-time packet forwarding approach relies on forwarding tables expanded with Fast Fourier Transform-based dual hashing to determine forwarding queries in OpenFlow [8].

B. Flow Classification and Steering

Network packets are encapsulated units of data transmitted over a network [9]. Their forwarding from the ingress to egress interfaces of a node is a fundamental function performed by routers and switches. Packet forwarding may be classified as either routing or switching, depending on whether a received packet traverses distinct IP address spaces for the source and destination addresses. Routing develops a sequence of rules and frameworks for guiding a packet from one node to another in such cases, whereas forwarding determines the next-hop interface that specifies the next-hop node. Routing operations typically generate forwarding tables and policies based on external information, often communicated through routing protocols. In contrast, medium-to-high forwarding rates and low processing latencies are important goals in switching, routing, and forwarding. Within the latter category, solution objectives often include support for various protocols and dynamic changes, which enter the realm of networking rather than purely switching.

Forwarding remains one of the primary functions performed by networking processors. The datapath of a typical forwarding engine consists of packet parsing, header inspection, flow classification, flow steering, and a number of other states. As packets traverse a forwarding engine, many decisions are made based on previously stored state information, typically with flow tables and other similar data structures. High packet throughput remains a focus for many applications, including mobile and cloud-based services, even as line rates enter the multi-Tb/s domain. Maintaining the packet-processing model, a number of forwarding functions require access to a state in the form of the packet's flow ID, destination, and/or other key attributes [10]. During the packet-processing pipeline, lookup and inspection decisions control packet flows through a forwarding datapath. The packet-processing model remains central to software-defined networking systems, permiting the installation of forwarding tables on switches by controlling the corresponding lookup state with control packets. The packet-processing model typically supports dynamic changes at packet rates determined by actual flow patterns. Within the forwarding engine, the well-defined data formats, header structures, and generally supported header fields permit significant flexibility in the choice of the specific architecture and subsequent implementation of flow classification and steering.

Flow classification is the machine learning (ML) function of determining the data-centric payload type that is transmitted as a stream from source to destination. Flow classification and the enumeration of the corresponding flow classification type can either elide or imply the need for subsequent flow steering. Flow classification can be thought of in terms of static lookup tables or classifiers, as well as functions that operate on character or byte sequences, in an effort to characterize a stream that occurs over longer periods. Classification of pseudorandom number sequences, s-flow, and similar techniques may enter into the discussion of flow classification [11].

The classification stage in a packet-processing workflow receives few distinct types of input. Frequently, a descriptor that identifies the predominant upper-layer protocol flows across active flows is enough information to implement the prioritization at the packet-triggered classification stage. The consideration of packet types—such as Transport Control Protocol (TCP) or User Datagram Protocol (UDP)—enables a straightforward classification of the upper-layer protocol. Various applications generate flows that aggregate to significantly higher data rates than legacy voice streams, and modem types continue to evolve. As the streaming becomes the key service feature, the upper-layer protocol at the packet-triggered classification stage frequently contains enough information to define simply and comprehensively the

upper-layer packet-processing protocol associated with that service. Even across distinctive packets, a network-layer packet-delay measurement reflects the extent of traffic forwarding and offers a comprehensive definition of a variety of packet-processing services.

Flow steering selects packet-processing states based on information stored in the per-flow (i.e., echo) flow table inside the control plane. Such selection typically follows classification and inspection decisions. Emulation of hash lookup-ensemble and ternary content addressable memory-state table (TCAM) technologies permits high-speed classification with multiple simultaneous hashes. Nonetheless, significant packets already traverse the packet-forwarding engine prior to state programming. This combination of considerations forms the basis for the variation of steering determination and selection.

C. Stateful Processing and State Machines

Forwarding performance can be significantly improved by executing the associated tasks on a GPU [12]. However, stateful GPU-accelerated tasks are challenging by nature: they require shared access to a potentially large amount of writable state; they exhibit irregular access patterns; and they often involve complex synchronization mechanisms.

The packets flowing through the network occupy several stages in typical forwarding systems, and forwarding states—flow entries, egress/output queues, and bandwidth per port—with stateful patterns are generated and utilized in each stage. Classification or steering is usually performed in the first stage after packet parsing and key extraction, determining the outgoing flow. Subsequent stages may refer to both active and passive per-flow statistics such as timeouts, packet counts, and byte counts, updating such statistics itself. The egress/input queue in the forwarding state indicates a forwarding flow.
Prioritization of certain classes through queues on a per-flow basis after forwarding decision is highly desired for quality of service (QoS), especially when shared medium is provided. The control plane queries the states (flow tables, statistics, queues) and modify them frequently—table updates are needed to install/delete flow entries, statistics are polled to observe forwarding behaviors, queue parameters are adjusted to limit queue size and temporally drop packets. Residing on the GPU with parallel processing on millions of packets, while the control-plane updates (reading, writing) are on a single concurrent—either running on the CPU or utilizing an additional and external device with limited parallelism. With the aid of hardware or API, shared writable states on the GPU from both datapath and control-plane are achievable although the frequently access and modification trade-off states (CPU/GPU) need to be reduced and designed carefully.

D. Quality of service and Traffic Shaping

To provide the best possible experience to users and to enable services requiring guaranteed performance, many applications on modern networks impose Quality of Service (QoS) constraints. QoS can be measured in terms of available bandwidth, end-to-end latency, jitter, reliability of transfer, and information loss, and appropriate guarantees can help satisfy service agreements in terms of desired performance. While most contemporary multipacket scheduling mechanisms in networking switch devices do not guarantee each packet the required QoS factors, GPU-based packet-processing applications on GPUs can offer different normative traffic-management operations, enabling the definition of schedules for packets with different types of service.

Traffic bursting, where packets of the same flow arrive in bursts separated by relatively long idle periods, is a common characteristic of real Internet traffic. In Internet Protocol (IP) networking, traffic shaping and traffic scheduling can thus be performed on packets in flows rather than on individual packets. Traffic shaping aims to limit the knowledge about the residual packet arrival process and is often achieved by flow buffer-size management. The shaping policy typically limits the number of packets sent until a certain amount of time after the previous packet departure. Various scheduling policies exist for

multiple flow packets of the same type, including First-Come First-Served (FCFS), Round Robin (RR), and Weighted Fair Queuing (WFQ). However, no mechanism determines the order of packets entering traffic shaping for a given output or the number of different typing classes that guarantee packets from one type will not enter the forwarding process before packets of another type [13].

V. SYSTEM ARCHITECTURE: CPU-GPU COLLABORATION

Traditionally, packet-processing tasks such as network address translation and rule lookups have been implemented on a dedicated application-specific integrated circuit (ASIC) chip; however, there are challenges in sustaining performance at multi-terabits per second (Tbps) line rates with evolving processes and persistent growth in network traffic. The potential of graphics processing units (GPUs) to simultaneously accelerate multiple packet-processing tasks, combined with broad adoption and continued high-volume production, makes the GPU an appealing candidate for auxiliary packet processing in general-purpose data-processing systems. Such acceleration still leaves the core, resource-intensive packet-processing tasks to commodity CPUs without necessitating a specialized processing engine. This approach offers significant advantages over fixed-function ASICs, including the ability to target broader applications, increased adaptability to changes in timing and content-delivery requirements, a more manageable upgrade path to emerging technologies, and the avoidance of vendor lock-in and allocation of complex verification resources.

Modern computing platforms often deploy CPUs and the associated complementary intellectual property blocks linked via a high-speed interconnect to avoid performance bottlenecks introduced by data transfers among these processors. Two distinct paradigms exist for packet-processing offload: the offload paradigm and the coprocessing paradigm, each influencing the organization of the system architecture and the modularity of the packet-processing functions. The offload approach completely transfers control of the packet-processing function from the host CPU to the packet-processing processor. Although this paradigm allows simultaneous operation of multiple processing elements, it necessitates a full reset of the processing function to restore control to the CPU. Further, the control-plane data path may function through a localized memory block. In contrast, the coprocessing paradigm enables both the CPU and the coprocessor to retain control of the same packet-processing function concurrently and facilitates more extensive control-plane data communication through a wider data path and additional addressing modes. The packet processor remains widely accessible without imposing a full context switch. The contemporary packet-forwarding processor—such as and similar—provides generic packet-deduplication capabilities that integrate smoothly into a CPU-centered architecture and follow the coprocessing design.

Two approaches are available for CPU–GPU communication: zero-copy communication, which eliminates explicit data transfer between CPU memory and GPU memory, and explicit data transfer, which transfers selected data between the host and the device. Zero-copy communication retains a unidirectional forward stream throughout and therefore remains transparent to the processing device in practice. Explicit data transfers can trigger interruptions on the processing units, causing marked delays. The CPU can assign packets requiring identical processing to separate queue pairs or distinct buffers to optimize transfer rates. The communication model offers resource management at the CPU level via more flexible configuration of buffer sizes. To minimize transfer overhead, packets enter the GPU processing if the queuing mechanism delivers few requests, while traffic on already fully populated queues remains invisible to the GPU.

Packet-processing functions typically exhibit cycle-level dependencies governed by separate control states, thereby demanding data forwarding at every packet arrival. Consequently, synchronization among multiprocessor groups must avoid global operations such as atomic instructions to enhance throughput. Programming environments such as OpenCL and CUDA provide support for per-resource atomic

operations with processor-group–level scope, reducing global synchronization requirements and improving device-level execution efficiency.

A. Offload versus Co-processing

Offload and coprocessing differ primarily in how the CPU and GPU collaborate. In the offload paradigm, the CPU causes the GPU to execute an application to completion. The forwarding engine appears as a simple stateless data path; the CPU controls flow classification and configuration while managing stateful processing operations. The GPU receives packets from, and returns them to, the same input-output (I/O) interface.

The coprocessing paradigm maintains a fully functional CPU-based forwarding engine alongside the GPU. Traffic is delivered to the CPU and then steered to the GPU for additional operations. Control messages also traverse this path. Both paradigms use the same control-plane logic. The choice between the two hinges on whether to implement flow classification on the CPU or to offload it entirely. The former configures the GPU according to the classification rules; the latter extracts application-independent information from packets, allowing the CPU to configure the packet-processing engine as needed [14].

B. Zero Copy and DMA Strategies

To achieve effective zero-copy transfers, the workstation and GPU must share a portion of the bus memory and use the same page table. Consequently, the GPU can directly access packets in the bus memory, eliminating the need to copy packets from host memory into a memory region reserved for the GPU. Instead, the CPU simply informs the GPU of newly received packets and their corresponding addresses in the bus memory. The GPU can then proceed to process packets without any additional copying. Maintaining the same page table between the CPU and the GPU also facilitates direct DMA transfers. When the GPU finishes processing packets, it can return them to the bus memory, thereby avoiding unnecessary transfers from the GPU memory. This CPU–GPU memory-sharing mechanism permits the implementation of network functions at both the CPU and GPU with overlapping data transfer and computation.

A DMA is a mechanism that allows hardware devices to transfer data to or from memory without processor intervention. Once set up, a DMA controller can operate independently and transfer data in parallel with the CPU. In GPU programming, a DMA can transfer data to or from the GPU device memory while the GPU is executing kernels [15]. Thus, the CPU and the GPU can simultaneously read and write to the bus memory, which considerably shortens the data-transfer time between the CPU and the GPU and reduces the overall packet-processing time.

C. Synchronization, Consistency and Concurrency Control

Packet forwarding applications executed on multiprocessors, such as GPUs, often need synchronization and concurrency control to ensure correct results. Highly parallel architectures impose stringent requirements, which many existing synchronization mechanisms do not meet. Confining accesses to shared data structures to well-defined, separate programming model constructs is an effective way to reduce or eliminate concurrency concerns. Programming models that rely on coarse-thread synchronization through external barriers, such as those used in allocating and releasing resources like packets or pipeline processing stages, expose concurrency but allow explicit handling of atomicity concerns. By adopting scenarios that do not require shared-memory synchronization, GPGPU applications can be mapped onto other heterogeneous architectures and recompiled for conventional processors.

Most multi-processor programming or multicore computer languages adhere to well-defined programming models for inter-processor operations in the presence of shared memory. The General-Purpose Computation on Graphics Processing Units (GPGPU) arena has witnessed the introduction of specialized programming models optimized for data-parallel computations on graphics hardware. Correspondingly, the existing multiprocessing approaches are proposed for multi-socket or shared-memory systems. The influence of the specific programming model on coordination principles, shared-data access delineation, and the scheduling of tasks in a multi-threaded application is of significant concern. Careful reforms and real-time re-evaluations of the implementation of packet-forwarding applications can help preserve the line between shared and non-shared resources [16].

VI. PERFORMANCE CONSIDERATIONS AND BENCHMARKING

Grappling with the limitations of conventional, CPU-centric approaches to packet-forwarding tasks, researchers have recently leveraged the architecture of Graphics Processing Units (GPUs)—massively parallel processing units initially developed for graphics processing—for packet forwarding in data networks. GPUs, particularly when integrated with programmable architectures that support general-purpose computation, have also been used in accelerating applications other than graphics and video rendering. As devices with a large number of computing cores, GPUs exhibit high throughput for parallel workloads of packet-forwarding tasks.

Their ability to process multiple packets concurrently allows systems employing GPUs to increase packet throughput while relying on programmable, generalized data-path hardware. Using this hardware, system designers can simultaneously modify applications to expand their functionality and/or incorporate new capabilities specific to packet forwarding. Applications written as sequences of individual instructions are partitioned into parallel “kernels,” each specified by a collection of simultaneity constraints.
While GPU hardware has the capacity to process application workloads in parallel, the effective bandwidth available for memory-intensive configurations is typically constrained to a single packet per couple of instruction cycles. Consequently, comprehensive parallelization of additional functionality, such as complicated inspections, extractions, time measurement, and modifications, is crucial for achieving higher throughput. Heterogeneous architectures, which consist of both GPU and CPU components, ensure packet-forwarding systems retain centralized control and minimizes the added complexity through GPU programming. GPUs alone are unable to perform the sequential control necessary for a complete network-packet-forwarding solution, which consists of both control and regular data planes.

GPU-enabled systems for packets forwarded in network environments still maintain various essential capabilities. With the architectural structure, parallelization remains an uncomplicated undertaking. Users can maintain control over multiple streams at once, including active packet counts, memory state, and connected ports. The limited temporal preservation inherent in DRAM technology remains compatible with conventional streaming packet-after-packet processes involving no elaborate internal state. Such capability permits the processing of additional, function-oriented data in conjunction with the original packets themselves. The workloads of commonplace applications are addressed without requiring alterations. Stream types characterized by wide-ranging data formats remain unencumbered while stateful or semi-stateful processing of generic formats becomes feasible within H.251 large-capacity stream framework.

A. Latency versus Throughput trade-offs

Nonblocking, low-latency forwarding architectures will deliver line-rate throughput with latency constrained to a few microseconds; emerging throughput-bound applications will enable full use of buffering and reduce throughput at most to hundreds of microseconds. Latency and throughput interactions depend on flow size and traffic patterns. Sources with finitely-sized packets require

independent state information—CPU-selected paths, hash tables, and CPU-consistent pointers—at the GPU's time of the last action. Throughput and 95th-percentile latency interact closely under small-packet, aggregate workloads; as overall throughput increases, latency nudges upward. Reducing the input packet set diminishes demand for independent state information, which license conduits to the GPU; linear, state-associated, and skipped-path hash lookups gain efficiency in such scenarios. Throughput and 95th-percentile latency remain positively correlated, though less pronouncedly, when the packet set's independence shrinks. Nevertheless, forwarding-on-four-ports architectures accrued, depending on the pattern, intermediate 95th-percentile latencies far below CPU-bound figures. Pipelined, multithreaded GPUs sustain small packet-set scenarios, though non-independence rebounds with more datacenters or links.

B. Bandwidth Utilization and Memory Bottlenecks

A GPU-accelerated packet-forwarding architecture can achieve data rates exceeding 100 Gbps with microsecond-level latencies at full line rate for typical traffic patterns. However, these performance metrics reveal several key performance limitations: bandwidth utilization and memory bottlenecks. Experimental results show that less than 20% of the available external device memory bandwidth is used; similar results are seen for PCIe transfers between a host CPU and a GPU. Profiling identifies packet-writing overheads, internal 128-bit transfers, and the final write to device memory as the bottlenecks.

Various levers exist for improving bandwidth utilization. Using shared-memory buffers on Fermi GPUs reduces the need for memory accesses by one or more transfers, considerably decreasing the effectiveness of internal 128-bit writes upon return to device memory. Reducing the frequency of 32-bit transfers would also provide significant benefits from the very high memory-bandwidth capacity of Fermi chips. Finally, newer devices with much larger on-chip shared memory and L1 caches permit access to packets already residing in programmable parsers for most messages. These attributes, especially on Kepler chips, alleviate some of CUDA C's obstruction to the throughput model, enabling the more expedient development of subsequent parallel packet-processing operations that further exploit fine-grained parallelism suitable for GPUs [17].

C. Energy Efficiency and Thermal Implications

Energy-efficient routing and forwarding depends on traffic patterns and average power consumption [18]. The performance on the leading GPU in a data-center-grade forwarding pipeline with representative traffic patterns demonstrates that power budgets remain far under the limits imposed by current server architecture. Excessive dynamic power indicates the importance of load-balancing strategies, traffic shaping, and queuing policies. A heat-flux analysis reveals that temperature increase caused by GPUs still falls within standard bounds for data centers. Across multiple workloads exhibiting diverse packet-processing requirements, a system delivering consistent levels of performance while cutting energy consumption is still feasible.

GPU-based forwarding systems benefit from a broadband interface, and the figures presented are nonetheless conditioned on lower-traffic flows. Concurrency and parallelism remain largely unexploited in options such as hash tables, flow classifiers, and address lookups, whose energy impact remains to be reduced through alternative algorithms or drastic reengineering.

VII. RECOMMENDED ARCHITECTURE OF A REAL APPLICATION OF A CPU – GPU PACKET FORWARDING CHIP

I propose a hybrid CPU–GPU packet forwarding engine derived from analysis of two reference documents on GPU-accelerated packet forwarding. The design focuses on batching, zero-copy/DMA transfers, and GPU-parallel fast-path processing while retaining CPU-based control-plane management.

A. High-Level Design Choice

The recommended architecture is a Hybrid CPU + GPU Coprocessing Model. The CPU manages NIC I/O, batching, scheduling, and control-plane updates, while the GPU accelerates data-plane stages such as parsing, lookup, and header rewriting.

B. Major Architectural Blocks

1) Ingress & CPU Front-End

NIC RX with multi-queue support, user-space packet I/O, batching, and packet descriptor creation.

2) Shared Buffers & Transfer Engine

Pinned host memory, zero-copy access or DMA transfers, and overlapped execution using double buffering.

3) GPU Fast Path Pipeline

Massively parallel kernels for parsing, classification (LPM/hash), header modification, and egress mapping.

4) Egress & Transmission

CPU or NIC-based queue management, scheduling, and batched packet transmission.

5) Control Plane

CPU-managed routing tables, flow rules, statistics, and synchronized GPU table updates.

C. GPU Fast Path Pipeline Stages

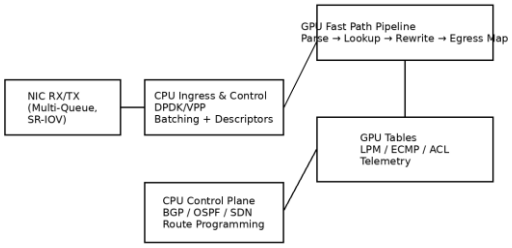
1. Parse & Key Extraction: Decode headers and extract lookup keys.
2. Classification & Lookup: Perform LPM routing, hash-based flow lookup, and ACL checks.
3. Action & Header Rewrite: Modify headers, decrement TTL, update checksums.
4. Egress Mapping: Assign packets to output ports and queues.

D. Logical Block Diagram (Textual)

NIC RX → CPU Batch Builder → Packet Descriptors → Shared Buffers → GPU Pipeline (Parse → Lookup → Rewrite → Egress Map) → CPU/NIC Scheduler → NIC TX

E. Key Design Principles

– Use batching to amortize GPU launch overhead.
- Overlap data transfer and computation.
- Minimize stateful operations in GPU kernels.
- Maintain CPU-based control-plane for flexibility and safety.



VIII. CONCLUSION

As a relatively recent area of research and development, GPU-accelerated packet forwarding offers significant opportunities. Multiple findings have identified portable, shared-memory programming models that enhance data-path parallelism and throughput without compromising packet-processing flexibility. Portable enhancements, such as single-pass packet classification through SIMD-compliant

data formats, simplify programming and improve performance across architectures [19]. Many unexplored avenues remain for additional performance improvements and deployment opportunities. To harness the full potential of packet-forwarding acceleration on GPUs, research can target portable architectures, programming models, workloads, and the integration of security, quality-of-service, and reliability features.

Efficient packet-forwarding on GPUs demands proper design and careful consideration of the targeted use cases. Many applications do not fully exploit the parallelism offered by packet processing, and high development costs hinder adoption. High-throughput packet forwarding remains an important but underexplored GPU application. Considerations such as programmability, ease of use, performance stability across different workloads and GPU generations, energy efficiency, and alignment with established standards can drastically improve the viability of GPU-based packet-forwarding solutions and relate closely to ongoing work in data-path programmability, heterogeneous computation, and in-network processing. Striking the appropriate balance among these factors will remain an open research question as the pressure to support ubiquitous connectivity and ever-increasing data rates mounts across the networking sector.

REFERENCES:

[1] M. Abbasi, R. Tahouri, and M. Rafiee, "Enhancing the performance of the aggregated bit vector algorithm in network packet classification using GPU," 2019. ncbi.nlm.nih.gov
[2] A. Lonardo, F. Ameli, R. Ammendola, A. Biagioni et al., "NaNet: a Low-Latency, Real-Time, Multi-Standard Network Interface Card with GPUDirect Features," 2014.
[3] C. Cascone, R. Bifulco, S. Pontarelli, and A. Capone, "Relaxing state-access constraints in stateful programmable data planes," 2018.
[4] C. L. Lee, Y. S. Lin, and Y. C. Chen, "A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection," 2015. ncbi.nlm.nih.gov
[5] S. Fuentes de Uña, "Analysis of Xilinx SDNet tool for packet filtering in 100 Gbps network monitoring applications," 2018.
[6] T. Wang, J. Lin, G. Antichi, A. Panda et al., "Application-Defined Receive Side Dispatching on the NIC," 2023.
[7] J. Meng, G. Nadeen, N. Ho-Cheung, C. Paolo et al., "Investigating the feasibility of FPGA-based network switches," 2019.
[8] S. Kassing, V. Dukic, C. Zhang, and A. Singla, "New primitives for bounded degradation in network service," 2022.
[9] D. Kim, N. Lazarev, T. Tracy, F. Siddique et al., "A Roadmap for Enabling a Future-Proof In-Network Computing Data Plane Ecosystem," 2021.
[10] X. Chen, J. Zhang, T. Fu, Y. Shen et al., "Demystifying Datapath Accelerator Enhanced Off-path SmartNIC," 2024.
[11] J. Meng, G. Nadeen, N. Ho-Cheung, C. Paolo et al., "Investigating the feasibility of FPGA-based network switches," 2019.
[12] X. Chen, J. Zhang, T. Fu, Y. Shen et al., "Demystifying Datapath Accelerator Enhanced Off-path SmartNIC," 2024.
[13] C. Cascone, R. Bifulco, S. Pontarelli, and A. Capone, "Relaxing state-access constraints in stateful programmable data planes," 2018.
[14] M. Abbasi, R. Tahouri, and M. Rafiee, "Enhancing the performance of the aggregated bit vector algorithm in network packet classification using GPU," 2019. ncbi.nlm.nih.gov
[15] I. Bhati, U. Dhawan, J. Gaur, S. Subramoney et al., "MARS: Memory Aware Reordered Source," 2018.
[16] V. E. N. K. A. T. RAMAN GOPALAKRISHNAN, "Packet Filtering Module For PFQ Packet Capturing Engine.," 2012.

[17] Y. Sun and Z. Guo, "The Design of a Dynamic Configurable Packet Parser Based on FPGA," 2023. ncbi.nlm.nih.gov
[18] D. Bermingham, Z. Liu, X. Wang, and B. Liu, "Field-based branch prediction for packet processing engines," 2009.
[19] L. McHale, P. V Gratz, and A. Sprintson, "Flow Correlator: A Flow Table Cache Management Strategy," 2023.
[20] M. L. Abbas, "Development of a multi-mode self-adaptive algorithm to create an efficient wireless network on a university campus," 2014.
[21] S. Rodhiah Mohd Yunus, "Quality Of Service (QOS) Improvement Using Traffic Shaping In ADTEC Batu Pahat," 2016.
[22] G. Salvador, W. H. Darvin, M. Huzaifa, J. Alsop et al., "Specializing Coherence, Consistency, and Push/Pull for GPU Graph Analytics," 2020.
[23] J. R. Alsop, "Specialization without complexity in heterogeneous memory systems," 2018.
[24] D. Alejandro Rivera-Polanco, "COLLECTIVE COMMUNICATION AND BARRIER SYNCHRONIZATION ON NVIDIA CUDA GPU," 2009. []
[25] M. Andersch, J. Lucas, M. Álvarez-Mesa, and B. Juurlink, "On latency in GPU throughput microarchitectures," 2015.
[26] G. Prekas, "Bridging the gap between dataplanes and commodity operating systems," 2018.
[27] B. C Johnstone, "Bandwidth Requirements of GPU Architectures," 2014.
[28] T. Hoang Vu, V. Cong Luc, N. Trung Quan, N. Huu Thanh et al., "Energy saving for OpenFlow switch on the NetFPGA platform based on queue engineering," 2015. ncbi.nlm.nih.gov
[29] D. Salopek and M. Mikuc, "Enhancing Mitigation of Volumetric DDoS Attacks: A Hybrid FPGA/Software Filtering Datapath," 2023. ncbi.nlm.nih.gov
[30] D. Fernando Bermúdez Garzón, C. Gómez Requena, P. Juan López Rodríguez, and M. Engracia Gómez Requena, "Speeding-up the fault-tolerance analysis of interconnection networks," 2015.
[31] L. Chen, G. Chen, J. Lingys, and K. Chen, "Programmable Switch as a Parallel Computing Device," 2018.