GPU-Accelerated IPv4 Packet Forwarding

Ferraris Luca

***Abstract*—Current IPv4 packet routing suffers from high per-packet processing costs, which can be a fundamental bottleneck in modern network environments.**
This bottleneck arises from the fact that the current networking stack is based on CPU forwarding for processing thousands of packets per second. The task involves a large number of independent and repetitive operations.
A workload that aligns well with the parallel execution model of modern GPU architectures.
This study draws inspiration from the PacketShader [1] project, focusing specifically on re-implementing its IPv4 forwarding component and extending it within an open-source framework. The proposed work develop a simplified but extensible IPv4 forwarding pipeline in CUDA, complemented by a baseline CPU implementation for comparative evaluation.

I. Introduction

NETWORK forwarding is the core part of the actual Internet infrastructure. Every packet traversing the network must be processed by one or more routers, which are responsible for inspecting packet headers and determining the appropriate forwarding path toward the destination. The traffic and the link speeds continue to grow, packet forwarding performance has become a critical factor.

In contemporary networking systems, packet forwarding is predominantly implemented in software running on general-purpose CPUs. Software routers are widely adopted due to their flexibility, and compatibility with virtualized and cloud-based environments. However, this architectural choice implies that forwarding operations are executed as a sequence of per-packet computations on the CPU, including header parsing, routing table lookup, and packet modification.

While CPUs provide strong single-thread performance and support multi-core parallelism, their architecture was not specifically designed for the highly repetitive and data-parallel nature of packet processing workloads. As a result, CPU-based forwarding pipelines are increasingly struggling to scale with increasing packet rates, especially in scenarios dominated by small packets and high throughput requirements.

These limitations motivate the exploration of alternative computing platforms capable of exploiting massive parallelism for packet processing. In this context, Graphics Processing Units (GPUs), offering thousands of lightweight threads well suited for network workloads.

A. IPv4 forwarding background

The IPv4 forwarding engine must parse the IP header, decrement the Time-To-Live (TTL) field, recompute the header checksum, and perform a routing table lookup (typically a Longest Prefix Match LPM) to determine the appropriate output interface. This operations are defined by IPv4 specifications (RFC 791 [2])

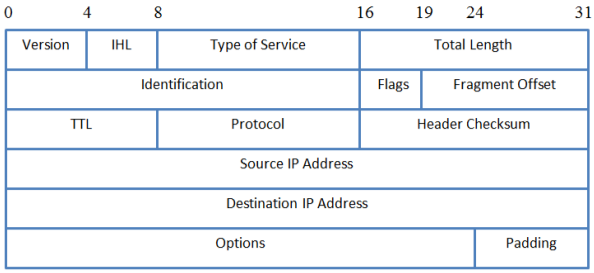

Fig. 1. Typical software router forwarding pipeline executed on a general-purpose CPU.

In modern router architectures, packet forwarding operations are implemented either in specialized hardware, such as ASICs and NPUs, or in software running on general-purpose CPUs. Software routers are becoming increasingly popular due to their flexibility, programmability, and reduced hardware development costs, and they are widely adopted in software-defined networking (SDN) and cloud environments.

B. Limits CPU-based

Modern CPUs offer high single-thread performance and multiple cores; however, they are not designed to efficiently execute thousands of identical operations in parallel, such as those required by IP packet forwarding. In high-speed networks, the dominant performance factor becomes the number of packets processed per second rather than raw bandwidth. If the CPU cannot perform all forwarding operations at a rate matching the link bandwidth, the router fails to sustain line-rate packet forwarding.

C. Benefits GPU-based approach

Graphics Processing Units (GPUs) provide massive data-parallel execution capabilities with thousands of lightweight threads designed for perform same operations on different data. This model matches the packet forwarding workload.

By offloading packet forwarding to the GPU it is possible to exploit fine-grained parallelism, amortize instructions overhead across large packet batches, use memory layout optimized for coalesced access, such as Structure-of-Arrays (SoA) to improve memory efficiency.

D. Target and Objectives

The primary objective of this work is the design and implementation of an open-source GPU-accelerated IPv4 packet forwarding, aimed at exploiting packet-level parallelism through the CUDA programming model. The forwarding pipeline includes the essential operations required by IPv4 routing, Time-To-Live (TTL) update, IPv4 header checksum recomputation

[3], and Longest Prefix Match (LPM) routing table lookup. To maximize memory efficiency on the GPU, packet data are organized using a Structure of Arrays (SoA) layout, enabling coalesced global memory accesses.

A second objective is the development of a clear and verifiable CPU-based reference implementation of the same IPv4 forwarding logic. This implementation serves as a functional baseline and enables a fair and reproducible comparison between CPU and GPU executions.

The project further aims to define and execute a comprehensive benchmarking methodology. Synthetic packet workloads are generated with varying batch sizes, ranging from 1 thousand to 1 million packets, in order to evaluate scalability. Additionally, routing table sizes are choose to be fix at 100 routes that permit analyze the impact of lookup complexity. The influence of packet characteristics is also studied by varying Time-To-Live distributions and payload sizes.

Performance evaluation is conducted across different GPU architectures, including a consumer-grade GPU (NVIDIA GTX 1060) and data-center-class GPUs available in the HPC@Polito infrastructure (NVIDIA A40 and NVIDIA V100). This analysis allows assessing the portability and scalability of the proposed approach across heterogeneous hardware platforms.

A detailed comparison between CPU and GPU implementations is performed. Specifically, the GPU forwarding kernel is evaluated against both a single-threaded CPU baseline and a multi-threaded CPU implementation based on OpenMP, highlighting the benefits and limitations of each execution model.

Finally, the performance of the system is quantified using a set of well-defined Key Performance Indicators (KPIs). These include packet throughput, expressed in millions of packets per second (Mpps) and gigabits per second (Gbps), batch processing latency, GPU resource utilization, and the speedup achieved with respect to CPU-based implementations.

II. CPU–GPU Comparison for Networking Tasks

A. Actual software router architecture

Software routers implement packet forwarding entirely in software on general-purpose CPUs, typically relying on the Linux networking stack or user-space frameworks such as DPDK or VPP. These systems follow a pipeline-based architecture in which packets traverse a sequence of processing stages, from packet reception to transmission.

The core operations of IPv4 forwarding are performed exclusively on packet headers, while the payload is usually left unmodified. This characteristic is particularly relevant for hardware acceleration, as it allows minimizing memory transfers and focusing computation on a small and fixed-size data structure. In traditional software routers, forwarding operations are applied on a per-packet basis, although modern implementations often process packets in batches to improve cache locality and throughput.

A typical software router forwarding pipeline consists of packet reception from the network interface, IPv4 header parsing, Time-To-Live (TTL) decrement and expiration check, IPv4 header checksum recomputation, Longest Prefix Match (LPM) routing table lookup, and packet transmission through the selected output interface. These stages are executed sequentially for each packet, while parallelism is exploited across packets using multi-core execution.

To achieve high performance, software routers adopt techniques such as polling-based packet reception, cache-aware data structures, and per-core routing tables. Nevertheless, packet processing remains constrained by memory bandwidth and limited instruction-level parallelism, especially when handling large volumes of packets at high data rates.

B. Proposed GPU stack

The proposed GPU stack implements IPv4 packet forwarding by offloading the core forwarding operations from the CPU to a CUDA-capable GPU. The design follows a batch-oriented execution model, where packets are grouped into large batches and processed in parallel on the GPU, exploiting packet-level parallelism.

Packet data are transferred from the host to the device using a Structure of Arrays (SoA) memory layout. Only the IPv4 header fields required for forwarding are stored and transferred, namely the destination IP address, Time-To-Live (TTL), and header checksum. The output interface index is stored in a separate array. This layout enables coalesced global memory accesses when consecutive GPU threads process consecutive packets, improving memory bandwidth utilization.

The forwarding logic is implemented as a CUDA kernel following a one-thread-per-packet execution model. Each GPU thread is responsible for processing a single packet and executes the same sequence of operations performed by the CPU reference implementation. Specifically, the kernel decrements the TTL field, recomputes the IPv4 header checksum, and performs a Longest Prefix Match (LPM) lookup on the routing table to determine the output interface. Packets whose TTL expires are dropped by marking the corresponding output interface as invalid.

The routing table is transferred to the GPU as a contiguous array of routing entries, each containing a network prefix, prefix length, and output interface identifier. For the baseline implementation, a linear-search LPM algorithm is adopted in order to ensure functional equivalence with the CPU reference and to simplify correctness validation. Although more advanced data structures could reduce lookup complexity, this approach provides a clear baseline for performance analysis (For example the Bloom filter data structures).

Kernel execution is configured using a one-dimensional grid of thread blocks, with each block containing a fixed number of threads. The total number of threads is chosen to match the batch size, ensuring that each packet is processed by a single GPU thread. Kernel execution time is measured using CUDA events, allowing accurate timing of the forwarding computation independently of host-side overheads.

This baseline GPU stack prioritizes correctness, clarity, and comparability with CPU implementations. It serves as a reference point for subsequent performance analysis, profiling, and optimization, including memory hierarchy tuning and architectural scalability evaluation.

C. GPU overheads

In GPU-accelerated packet processing, one of the primary performance challenges is the overhead associated with data transfers between the host (CPU) and the device (GPU). Unlike CPU-based implementations, GPU kernels operate on data residing in device memory, requiring explicit memory management and data movement through the PCIe interconnect. As a result, memory transfer overhead can significantly impact the overall performance of GPU-based forwarding pipelines.

In CUDA programming, the most straightforward approach to device memory allocation is the `cudaMalloc()` function, which allocates memory in the GPU global memory space. When data resides in standard pageable host memory, transfers between host and device require an intermediate staging step, where the CUDA runtime first copies data into a temporary page-locked buffer before initiating the Direct Memory Access (DMA) transfer to the GPU. This additional copy introduces latency and limits achievable transfer bandwidth, making pageable memory suboptimal for high-throughput workloads.

Furthermore, memory allocated using `cudaMalloc()` is not directly accessible by the host CPU, and any access requires explicit `cudaMemcpy()` operations. In batch-oriented packet processing, the cumulative cost of these memory transfers can become a dominant factor, potentially outweighing the benefits of GPU parallelism. Synchronization between host and device further contributes to execution overhead, especially in tightly coupled processing pipelines (the `cudaMemcpy()` implicit sync the threads).

In addition, GPU kernel invocation introduces a fixed overhead that becomes non-negligible for small batch sizes. Consequently, kernel launches should be amortized over sufficiently large workloads, or alternatively avoided by using continuously running kernels.

To mitigate memory transfer overheads, this project adopts CUDA pinned (page-locked) host memory allocations. Pinned memory prevents the operating system from paging the allocated memory to disk, allowing the GPU to directly access host memory through DMA. As a result, pinned memory enables higher transfer bandwidth and lower latency compared to pageable memory, reducing host-to-device and device-to-host transfer overheads and improving overall throughput.

Overall, these overheads highlight that GPU acceleration is beneficial only when data transfer and execution costs are carefully optimized and amortized over sufficiently large workloads.

III. System design and architecture

A. Architectural Overview

The system implements an IPv4 forwarding engine supporting both CPU and GPU execution paths, with the goal of evaluating the benefits of GPU acceleration for networking workloads. The architecture separates data representation, routing logic, and execution platforms (CPU and CUDA environment), enabling a fair performance comparison between different implementation strategies.

Two core data structures are used to represent the forwarding state: the packet data structure and the routing table entry structure.

Due to the different execution and memory access models of CPUs and GPUs, packet data must be represented differently in host and device memory. GPU execution relies on Single Instruction, Multiple Data (SIMD) parallelism and requires coalesced global memory accesses to achieve high performance. As a result, distinct memory layouts are adopted for the host and device.

On the host side, packet data are stored using an Array of Structures (AoS) layout, where each packet is represented as a single structure. This representation is convenient for packet manipulation and closely resembles the data layout used in traditional software routers. On the device side, packet data are reorganized using a Structure of Arrays (SoA) layout, which allows consecutive GPU threads to access contiguous memory locations, enabling efficient coalesced memory accesses.

The SoA packet representation used by the GPU implementation is shown in Listing 1. Each array stores a single IPv4 header field for all packets in a batch, while the output interface is stored separately.

Listing 1. Structure of Arrays (SoA) packet representation used on the GPU

```
struct PacketSoA {
    // Destination IPv4 addresses
    std::vector<uint32_t> dst_ip;
    // Time-To-Live (TTL) values
    std::vector<uint8_t>  ttl;
    // IPv4 header checksums
    std::vector<uint16_t> hdr_checksum;
    // Output interface indices
    std::vector<int>      out_if;
};
```

The routing table is represented as an array of routing entries Listing 2, each containing a network prefix, its prefix length, and the corresponding output interface. This representation is shared between the CPU and GPU implementations to ensure functional equivalence.

Listing 2. Routing table entry representation

```
struct RouteEntry {
    // Network prefix
    uint32_t prefix;
    // Prefix length (0--32)
    uint8_t  prefix_len;
    // Output interface index
    int      out_if;
};
```

The system integrates comprehensive timing instrumentation to measure end-to-end execution latency for each implementation variant. For CPU implementations, measurements capture pure computation time. For GPU implementations, measurements include data structure conversion (AoS-to-SoA), host-to-device transfers, kernel execution, and device-to-host transfers, providing realistic performance.

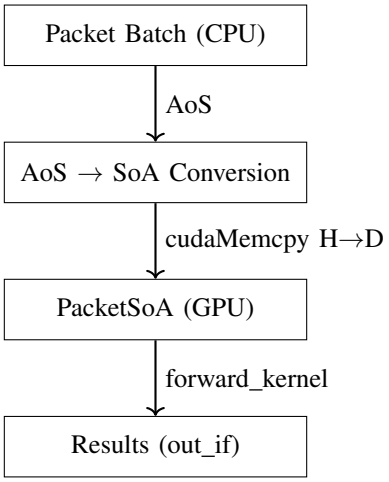

Fig. 2. CPU–GPU IPv4 forwarding architecture and data flow.

A basic validation framework verifies functional correctness by comparing output interface assignments across all implementations for identical input packet sets.

Figure 2 summarizes the overall CPU–GPU data flow and execution model adopted in the proposed architecture.

B. Implementation strategy

The architecture implements three distinct execution strategies to evaluate parallelization benefits. The CPU baseline implementation provides a sequential reference executing the complete pipeline for each packet in program order. The CPU parallel implementation leverages OpenMP to distribute packet processing across multiple CPU cores, maintaining the same algorithmic approach while exploiting thread-level parallelism.

The GPU implementation parallelizes processing at the packet level, assigning one CUDA thread per packet to execute the forwarding pipeline independently, maximizing throughput through massive parallelism.

An advanced GPU Bloom filter variant introduces a probabilistic pre-filtering stage to reduce routing table lookup overhead. This optimization employs a Bloom filter to quickly eliminate impossible route matches before performing the complete LPM algorithm. However, this solution does not scale well for large data sets but do good work for small data sets.

The evaluation framework requires a controlled and reproducible packet dataset to ensure consistent performance measurements across the different implementation. The system generates synthetic IPv4 packets programmatically through a specific function (that create RFC 791 compliant packets) with randomized destination addresses between the available one.

C. CPU implementation

A critical design consideration in CPU-based packet forwarding is byte order management between network representation and host processing. The IPv4 protocol mandates big-endian byte order (most significant byte first) for all multi-byte header fields, as specified in RFC 791. However, modern x86/x64 architectures natively use little-endian byte order. Consequently, two conversion functions are required to translate data between network-to-host and host-to-network representations. These conversions must be carefully handled during packet processing to ensure correctness.

1) CPU baseline implementation: The CPU baseline implementation provides a reference sequential execution model in which packets are processed individually. The core forwarding function follows a four-stage pipeline for each packet: IPv4 header validation, TTL validation and decrement, checksum validation (after which the network-to-host byte order conversion is performed), and finally Longest Prefix Match (LPM) routing table lookup. This implementation serves as the correctness baseline against which all optimized versions are compared.

The CPU implementation adopts an Array-of-Structures (AoS) memory layout, where each packet's data is stored contiguously in memory.

The sequential implementation exhibits predictable performance characteristics. Its time complexity is $O(N \times M)$, where N is the number of packets and M is the routing table size; however, this approach does not scale to high packet rates.

2) CPU parallel implementation: The parallel CPU implementation extends the baseline design by exploiting multi-core processors through OpenMP-based parallelization. The `#pragma omp parallel for` directive distributes packets across available CPU threads using static scheduling, ensuring balanced workload distribution. Each thread processes a contiguous chunk of the packet array, preserving cache locality within the thread's working set. The routing table is shared among threads in a read-only fashion, eliminating the need for synchronization. Despite these optimizations, CPU-based parallelization remains fundamentally limited by core count and memory bandwidth, motivating the exploration of massively parallel architectures such as GPUs.

D. GPU implementation

The GPU implementation exploits massive data-level parallelism inherent in packet forwarding by assigning one CUDA thread per packet. This follow part describes the data structure transformation, kernel design, thread configuration strategy, and the Bloom filter optimization variant.

1) Structure-of-Arrays Data Transformation: A fundamental prerequisite for efficient GPU execution is converting packet data from the CPU's Array-of-Structures (AoS) layout to a Structure-of-Arrays (SoA) representation. This transformation is critical for achieving coalesced memory accesses on GPU architectures, where adjacent threads accessing adjacent memory locations can merge into a single memory transaction.

The main motivation for this choice is that the GPUs organize threads into warps (32 threads on NVIDIA architectures) that execute instructions in lockstep. When thread 0 accesses `dst_ip[0]` , thread 1 accesses `dst_ip[1]` , and so forth, the memory controller can coalesce these requests into a single 128-byte transaction. With AoS layout, each thread would access different offsets within scattered packet structures, resulting in uncoalesced accesses and memory bandwidth degradation, so it is mandatory to carry out this transformation. In addition this transformation is done at host side and during this operation is done also the network-to-host transaction.

2) GPU Kernel Design: The GPU kernel that operate the forwarding operation is done to have a pipeline that process one packet per thread (using a 1-dimension array). No inter-thread communication is needed from the moment that the packets are not related each other

3) Thread Configuration Strategy: The kernel launch configuration determines the degree of parallelism through block and grid dimensions, directly impacting GPU occupancy and performance. **The implementation adopts a fixed block size of 256 threads per block**, a design choice resulting from architectural analysis and empirical validation across multiple GPU generations.

The choice of 256 threads per block is motivated by several architectural considerations that balance resource utilization, memory latency hiding, and hardware constraints. A block size of 256 ensures perfect alignment with 8 full warps, eliminating inefficiencies that arise from partial warp execution where some threads remain idle while others work.

Memory latency hiding represents a critical performance factor for this memory-bound workload. Global memory accesses to routing table entries and packet data incur latencies of 200–400 clock cycles. By maintaining thousands of active threads across the GPU (256 threads/block multiplied by tens of thousands of blocks), the hardware thread scheduler can rapidly context-switch between warps, ensuring that while some threads wait for memory operations, others continue executing arithmetic and control flow instructions. This massive parallelism transforms memory latency from a performance bottleneck into a hidden cost.

The decision to adopt a static configuration reflects a pragmatic engineering trade-off: sacrificing theoretical flexibility for demonstrable simplicity and predictability.

4) Memory Transfer Pipeline: GPU execution inherently involves orchestrating data movement between host and device memory spaces, a process that encompasses device memory allocation, bidirectional data transfers, kernel execution, and resource cleanup. The architectural design of this memory management pipeline significantly impacts both performance and code complexity. This implementation explores two distinct strategies: **standard pageable memory** and **pinned memory**

5) Standard Memory Allocation: The default memory allocation mechanism in CUDA is based on the use of `cudaMalloc()` for device memory and standard pageable memory on the host. When host data resides in pageable memory, the operating system retains the ability to relocate or swap memory pages, preventing the GPU from directly accessing host memory through Direct Memory Access (DMA).

As a consequence, host-to-device and device-to-host memory transfers involving pageable memory require an intermediate staging step managed by the CUDA runtime. During a transfer, data is first copied from pageable host memory into an internal page-locked buffer, after which the GPU performs the DMA transfer to device memory. This additional copy introduces extra latency and reduces effective bandwidth, making standard pageable memory unsuitable for high-throughput data transfer scenarios.

Furthermore, memory allocated with `cudaMalloc()` is accessible exclusively by the GPU, requiring explicit `cudaMemcpy()` operations for any interaction with host code. In batch-oriented packet processing pipelines, the cumulative overhead of these transfers can significantly impact overall performance and limit scalability. So this solution is good for a single batch implementation for the low overhead, but for the multi-batch processing aren't the best choice.

6) Pinned Memory Concept and Advantages: Pinned memory, allocated through CUDA-specific functions like `cudaHostAlloc()` or `cudaMallocHost()`, is locked into physical RAM locations by the operating system. This page-locking prevents the OS from swapping these pages to disk, ensuring that their physical addresses remain constant. This stability enables the GPU memory controller to perform direct DMA transfers without intermediate staging buffers, eliminating one complete memory copy operation from the transfer path.

The performance advantages of pinned memory manifest in two primary ways. First, eliminating the staging buffer reduces transfer latency. Second, pinned memory enables asynchronous transfer operations via `cudaMemcpyAsync()`, which can overlap data transfers with kernel execution or other operations, potentially hiding transfer latency entirely in pipeline-parallel execution models.

7) Bloom Filter Optimization Variant: The standard Longest Prefix Match implementation employs a linear scan of the routing table for each packet, an approach whose computational cost scales linearly with routing table size. While this strategy performs adequately for small routing tables (fewer than 20 entries), it becomes increasingly prohibitive as routing complexity grows. To address this scalability limitation, an advanced variant incorporates a **Bloom filter**—a

space-efficient probabilistic data structure—to prune the search space by quickly eliminating impossible route matches before performing exhaustive verification.

Bloom filters operate as probabilistic set membership testers with a critical asymmetric property: they produce **zero false negatives** (if the filter indicates an element is not in the set, this answer is guaranteed correct) but tolerate controlled **false positives** (if the filter indicates presence, verification is required). This property makes Bloom filters ideal for routing table acceleration: they can safely eliminate route candidates without risking incorrect packet drops, while occasional false positives merely result in unnecessary verification checks that produce the correct answer.

8) Bloom Filter Construction and Hash Design: The Bloom filter is constructed during a preprocessing phase on the CPU before kernel launch, using a 1-megabit (2^{20} bits, occupying 32 KB) bit array. Each routing prefix is inserted into the filter using two independent hash functions.

The implementation employs a packed bit representation where the 2^{20}-bit filter is stored as an array of 32,768 32-bit words. Bit manipulation operations (right shift by 5 for word index, bitwise AND with 31 for bit offset) enable efficient bit setting and testing without division operations.

9) Bloom-Accelerated Kernel Algorithm: The Bloom filter integration fundamentally alters the routing algorithm's search strategy. Rather than scanning the routing table linearly for each packet, the Bloom-accelerated kernel iterates through all possible prefix lengths (from 32 bits down to 0) in descending order, checking each candidate network prefix against the Bloom filter before consulting the routing table:

The iteration proceeds from longest to shortest prefix, respecting the Longest Prefix Match semantic where more specific routes take precedence over less specific ones. For each prefix length, the algorithm masks the destination IP to create a candidate network address, tests it against the Bloom filter, and only performs routing table verification if the Bloom test passes.

If either hash function indicates a missing bit, the candidate prefix is guaranteed not to exist in the routing table, enabling immediate continuation to the next prefix length without expensive routing table traversal.

IV. Experimental evaluation

This section presents an experimental evaluation of the proposed GPU-based IPv4 forwarding pipeline. The goal of the evaluation is to quantify the performance benefits and overheads of GPU acceleration compared to CPU-based implementations, with particular focus on packet throughput, memory transfer costs, and the impact of different memory allocation strategies.

All experiments compare three implementations: a sequential CPU baseline, a multi-threaded CPU implementation using OpenMP, and multiple GPU-based forwarding variants implemented in CUDA. Measurements are conducted under controlled conditions using synthetic packet workloads and configurable routing table sizes, allowing a detailed analysis of scalability and performance trade-offs.

A. Experimental setup

Experiments were conducted on a system equipped with an Intel Core i5-14400F CPU and an NVIDIA GeForce GTX 1060 GPU. The CPU features multiple cores and supports hardware multithreading, while the GPU provides a massively parallel execution model based on NVIDIA CUDA.

The software environment consists of a C++ implementation for the CPU-based forwarding pipeline and a CUDA-based implementation for GPU acceleration. The parallel CPU version uses OpenMP to exploit multi-core parallelism. GPU experiments were performed using both standard pageable memory and pinned (page-locked) host memory in order to evaluate the impact of memory allocation strategies on data transfer overhead.

B. Packet dataset and routing table

Synthetic IPv4 packet batches are generated to emulate realistic forwarding workloads. Each packet contains a valid IPv4 header with randomized destination addresses, allowing controlled evaluation of routing table lookups. Experiments were performed using batches of up to 10 million packets.

The routing table consists of 101 entries and supports Longest Prefix Match (LPM) lookups. In addition to the standard LPM implementation, an optimized variant using a Bloom filter is evaluated to reduce unnecessary routing table accesses. The routing table is shared across all implementations to ensure fair comparison.

C. Results

1) CPU results: The CPU baseline implementation processes packets sequentially and serves as a correctness and performance reference. For a batch size of 10 million packets, the sequential CPU implementation requires 2252.67 ms to complete the forwarding process.

The parallel CPU implementation significantly improves performance by exploiting multi-core parallelism through OpenMP. Under the same workload, the parallel version reduces the forwarding time to 685.09 ms, achieving a speedup of approximately 3.3× compared to the sequential baseline. Despite this improvement, CPU performance remains constrained by limited parallelism and per-packet processing overhead.

TABLE I
CPU FORWARDING PERFORMANCE

Packets (N)	CPU Seq (ms)	CPU Parallel (ms)
10^4	42.09	18.38
10^5	455.75	73.02
10^6	2252.67	685.08

2) GPU results: GPU-based forwarding performance was evaluated using both standard pageable memory and pinned host memory allocations. The execution time is decomposed into host-to-device transfers, kernel execution, and device-to-host transfers.

When using pinned memory, the GPU kernel execution time remains approximately constant at around 22 ms for the LPM

TABLE II
GPU FORWARDING LPM TIME BREAKDOWN (PINNED = NO)

Packets (N)	H2D (ms)	Kernel (ms)	D2H (ms)	Total (ms)
10^4	0.18	0.26	0.13	0.58
10^5	1.20	1.82	0.71	3.74
10^6	9.12	20.45	5.76	35.35

implementation and 21 ms for the Bloom-based variant. Host-to-device and device-to-host transfers require approximately 5 or 8 ms each, resulting in a total GPU execution time of roughly 35 ms per batch.

However, additional preprocessing overheads significantly affect end-to-end performance. In particular, the Array-of-Structures to Structure-of-Arrays (AoS-to-SoA) conversion and pinned memory allocation introduce substantial costs, reaching up to 409 ms for the LPM variant and 233 ms for the Bloom-based variant. These overheads dominate the total GPU forwarding time for single-batch execution.

D. Comparison and Analysis

For single-batch execution, GPU-based forwarding does not outperform the parallel CPU implementation due to substantial preprocessing and memory allocation overheads. In particular, the AoS-to-SoA conversion and pinned memory allocation costs outweigh the benefits of fast kernel execution.

However, multi-batch experiments reveal a different performance profile. When pinned memory allocations are reused across multiple batches, total execution time is reduced by approximately 60%, achieving an average processing time of 41.96 ms per batch. In contrast, repeated pageable memory allocations result in significantly higher overhead, with an average of 167.73 ms per batch.

TABLE III
MULTI-BATCH GPU PERFORMANCE

Memory type	Batches	Total time (ms)	Avg per batch (ms)
Pinned	10	672.61	41.95
Pageable	10	1677.73	167.73

These results demonstrate that GPU acceleration becomes advantageous only when overheads are amortized over multiple batches or continuous workloads. This behavior highlights the importance of memory reuse and streaming-oriented execution models for achieving high throughput in GPU-based packet forwarding systems.

This result confirms that GPU acceleration is workload-dependent and must be carefully integrated into streaming-oriented forwarding pipelines.

V. HPC EVALUATION AND PROFILING

This section reports the experimental evaluation performed on the HPC@PoliTO GPU infrastructure and presents a profiling-driven analysis of the forwarding pipeline. Results are compared against the local workstation measurements (GTX1060) to highlight the impact of compute capability and, more importantly, end-to-end overheads (memory transfers, allocations, synchronization, and host-side preprocessing).

TABLE IV
SLURM JOB CONFIGURATION AND PROFILING COMMAND (HPC RUN).

Partition	`gpu_v100`
Nodes	1
Tasks per node	4
GPUs requested	4 (`--gres=gpu:4`)
CPUs per task	8 (`--cpus-per-task=8`)
Memory	16 GB (`--mem=16G`)
Walltime	00:30:00
Modules	`gcc/12.4.0`, `nvhpc/25.1`, `openmpi/5.0.7`
Profiler	`nsys profile --trace=cuda,osrt,mpi,nvtx --sample=cpu`
Launch	`srun -n4 --cpu-bind=cores ./gpu_router`

TABLE V
PER-RANK EXECUTION TIME ON HPC (10M PACKETS, 101 ROUTES).

Implementation	Time [ms]	Throughput [Mpps]	Speedup
CPU sequential	2041.38	4.90	1.00
CPU OpenMP	2033.19	4.92	1.00
GPU LPM (pinned)	647.31	15.45	3.14
GPU LPM (pageable)	137.89	72.52	14.74
GPU Bloom (pinned)	216.19	46.26	9.40
GPU Bloom (pageable)	138.26	72.33	14.71

A. HPC job configuration and software environment

Experiments were executed through SLURM on the `gpu_v100` partition, requesting one node with four GPUs. The application was launched with four tasks (one per GPU) and CPU core binding enabled to reduce OS scheduling noise. Profiling was performed with Nsight Systems (`nsys`) using CUDA [4], OS runtime, MPI [5], and NVTX traces, plus CPU sampling.

B. Profiling methodology

Nsight Systems was used to attribute runtime across (i) CUDA API overheads (memcpy, allocations, synchronizations), (ii) GPU kernel execution, and (iii) memory transfer activity (Host-to-Device and Device-to-Host). The generated report did not contain NVTX ranges; therefore, phase segmentation (AoS→SoA, H2D, kernel, D2H) relies on API/memop attribution rather than explicit user-defined ranges. Adding NVTX markers around pipeline stages is recommended to obtain an unambiguous end-to-end phase breakdown.

C. HPC performance results (single-shot execution)

The profiled run used **10 million packets**, a routing table of **101 entries** (100 routes plus default), and tested both LPM and Bloom-based GPU forwarding. Since the application was launched with four tasks, console output contains per-task measurements; Table V reports the per-rank mean and standard deviation across the available prints.

a) Observation.: In a single-shot configuration, pinned memory can be counterproductive if page-locked allocations (`cudaHostAlloc`/`cudaFreeHost`) are performed repeatedly or are coupled with host-side preprocessing. This motivates a streaming-oriented design where pinned buffers are allocated once and reused across batches.

TABLE VI
MULTI-BATCH EXPERIMENT (10 BATCHES): PINNED REUSE VS PAGEABLE ALLOCATIONS.

Strategy	Total time [ms]	Avg per batch [ms]	Improvement
Pageable	1355.03	135.50	≈58%
Pinned	561.80	37.42	baseline

TABLE VII
CUDA API CALLS

CUDA API	Time share [%]	Total time [ms]	Calls
cudaMemcpy	44.5	2147.16	336
cudaHostAlloc	37.1	1790.49	224
cudaFreeHost	5.7	276.20	224
cudaStreamSynchronize	5.6	268.23	324
cudaEventSynchronize	3.2	153.40	288
cudaMalloc	1.6	78.11	656

TABLE VIII
GPU KERNEL SUMMARY

Kernel	Instances	Avg time [ms]	Total time [ms]
forward_kernel_bloom	88	1.33	117.46
forward_kernel	8	1.53	12.21

D. Streaming-oriented experiment (multi-batch buffer reuse)

To emulate a streaming pipeline, a multi-batch test processes 10 consecutive batches. Pinned buffers are allocated once and reused, while the pageable baseline allocates per batch. This configuration isolates the benefit of buffer reuse and amortizes allocation costs.

E. Nsight Systems results

Profiling confirms that end-to-end performance is strongly impacted by host-side overheads and memory operations.

1) CUDA API overheads: The CUDA API summary shows that a significant portion of traced time is spent in memory transfers and page-locked host allocations. In particular, `cudaMemcpy` and `cudaHostAlloc` dominate the CUDA API time.

2) GPU kernel execution: Kernel execution on V100 is relatively small compared to the observed API and transfer overheads.

3) Memcpy activity (time and size): The MemOps summaries indicate a large number of H2D/D2H transfers, consistent with a fragmented transfer pattern (multiple arrays / multiple copies). Consolidating transfers or generating SoA directly can reduce the number of memcopies.

F. Comparison with local workstation results

Local measurements (GTX1060) on the same workload show that end-to-end GPU performance is still bounded by host-side preprocessing and transfer orchestration. Table IX reports the best local configuration (pageable buffers to avoid allocation overhead in the single-shot case) and compares it to the HPC per-rank results.

TABLE IX
LOCAL (GTX1060) VS HPC (V100) PAGEBLE

Platform	CPU [ms]	GPU LPM [ms]	GPU Bloom [ms]
GTX1060	640.66	201.79	216.02
HPC (V100)	2033.19	137.89	138.26

G. Discussion

The HPC profiling confirms that kernel execution is not the dominant component on V100: compute is fast, but the end-to-end pipeline is largely governed by memory transfers, host-side allocations, and synchronization. The multi-batch experiment demonstrates that pinned memory becomes advantageous when buffers are reused across batches, as allocation overhead is amortized and throughput becomes limited by sustained transfer bandwidth.

In practice, beating a highly optimized CPU OpenMP baseline requires: (i) reusing pinned buffers (avoid repeated `cudaHostAlloc`/`cudaFreeHost`), (ii) reducing the number of memcopies (e.g., fewer/larger transfers or direct SoA generation), and (iii) minimizing synchronization points to allow overlap of transfers and kernels.

VI. CONCLUSION AND FUTURE WORK

Code Availability

The source code and scripts to reproduce the experiments are available at:
https://github.com/LucaFerro01/GPURoutingProject.

This work presented an open-source GPU-accelerated IPv4 forwarding pipeline in CUDA. Experiments on a local workstation (GTX1060) and on an HPC platform (V100) show that GPU kernels can execute the forwarding logic efficiently, but end-to-end performance is strongly dominated by host-side pre-processing, memory allocations, and PCIe transfers. In single-shot execution, these overheads can offset the benefit of GPU parallelism, making a well-optimized CPU OpenMP baseline competitive. Conversely, in streaming-oriented configurations where buffers are reused across batches, GPU execution becomes advantageous because allocation costs are amortized and transfers approach sustained bandwidth. Profiling with Nsight Systems confirms that a large fraction of runtime is spent in cudaMemcpy and page-locked host allocations, while kernel time remains comparatively small on V100. These results indicate that beating multi-core CPU forwarding requires a pipeline design that minimizes transfers and synchronization, reuses pinned buffers, and reduces host-side data transformations.

Future work will focus on eliminating AoS→SoA conversion overhead by generating SoA packets directly, overlapping H2D/D2H transfers with kernel execution using asynchronous copies and streams, and adopting more scalable routing-table data structures on the GPU to handle large tables efficiently.

REFERENCES

[1] K. Jang, S. Han, S. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," pp. 195–206, 2010. [Online]. Available: https://keonjang.github.io/papers/sigcomm10ps.pdf

[2] "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc791

[3] A. Rijsinghani, "Computation of the Internet Checksum via Incremental Update," RFC 1624, May 1994. [Online]. Available: https://www.rfc-editor.org/info/rfc1624

[4] "Nvidia cuda compiler driver nvcc," NVIDIA, Dec. 2025, accessed: 2026-01-23. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf

[5] "Openmp application programming interface, version 6.0," OpenMP Architecture Review Board, Nov. 2024, accessed: 2026-01-23. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf