CPU Scheduling Optimized Vectraflow

Evan Li
Brown University
evan_li1@brown.edu

Abstract

The integration of Large Language Models (LLMs) into data processing pipelines has become increasingly common, but it comes with performance challenges and often times, the bottle neck is GPU inference times. This work investigates the impact of client-side CPU resource management on end-to-end LLM pipeline performance, even when the GPU is the primary bottleneck. We explore methodologies for optimizing client-side execution by evaluating the effects of simulated CPU load and explicit CPU scheduling techniques (core pinning and process prioritization) within a Python-based data pipeline interacting with a vLLM server. Our experiments demonstrate that while default OS schedulers can manage client resources adequately under low system load, targeted client-side CPU scheduling yields some improvements in throughput and latency (particularly tail latency and handoff delays) when the GPU is constrained and client CPU contention is high. This suggests a non-linear relationship where the efficacy of such client-side optimizations becomes more pronounced under systemic pressure, highlighting the importance of holistic pipeline tuning.

1 Background

The data processing landscape has been evolving driven both by the explosion of complex data types (especially vector embeddings) [1] and the way LLMs are being integrated into analytical workflows [2]. Our work builds on the principles and architecture of data flow systems designed for these modern workloads, particularly systems like VectraFlow [3].

VectraFlow was originally conceptualized as a stream-oriented data flow engine for applications needing continuous, low-latency processing of vector data [3]. While we first explored these ideas in C++ (as noted in the background of earlier work [3]), we've since moved our implementation over to Python. This wasn't just a random choice, Python gives us a much ecosystem for LLM integration, lets us define complex data transformations more easily, and frankly, makes it faster to prototype pipelines that need to call LLMs frequently.

Traditional data processing pipelines typically implement operations you'd find in relational databases - your standard joins, filters, aggregations, and group-bys, concepts underpinned by general operating system and data management principles [4, 5]. What VectraFlow does is extend this approach by directly supporting vector data types and introducing specialized vector-based operators [3]. These include V-Filter, iV-Filter (inverse vector filter), V-TopK, iV-TopK (inverse vector top-k), and V-Join [3]. We need these operators for applications that require semantic understanding and similarity-based operations on streaming data - things like catching copyright violations in real-time or continuously evaluating LLM prompts, as discussed as use cases for VectraFlow [3]. The iV-Filter and iV-TopK operations are particularly unique to streaming contexts, where base vectors essentially act as continuous queries against incoming data streams.

Our research uses this Python-based pipeline framework to tackle key aspects of modern AI-driven applications. We've spent considerable time on the interaction between these pipelines and LLMs, building guardrails to ensure outputs are safe and reliable, and creating multi-stage pipelines for tasks like processing academic papers from arXiv. Getting these pipelines to run efficiently, especially when LLMs are involved, has been crucial.

One of the big challenges we've faced with LLM-integrated pipelines is performance optimization, a topic also central to efficient LLM serving [2]. Everyone typically points to the GPU that runs LLM inference as the main bottleneck. This got us wondering: even when the GPU is clearly the end-to-end bottleneck, can tweaking client-side CPU resource management and scheduling still give us meaningful performance gains?

Modern operating systems like Linux with its Completely Fair Scheduler (CFS) already do a decent job managing CPU resources [4, 5]. But our high-throughput, low-latency pipelines, especially at those critical "handoff" points where client-prepared data gets sent to GPU-accelerated LLM servers—might benefit from more explicit CPU scheduling strategies, building on fundamental OS concepts [4, 5]. Things like cache locality, NUMA awareness, and making sure critical threads get immediate CPU core access could make a real difference.

In this research, we've investigated how client-side CPU load (which we simulate through a CpuStressOp) and various CPU scheduling techniques (including core pinning with taskset and priority adjustment with nice) affect end-to-end performance across different pipeline configurations. We wanted to understand exactly when these client-side optimizations become most useful, particularly when the GPU is maxed out, and figure out which pipeline stages are most affected by CPU scheduling interventions. Our goal has been to provide practical insights that help build more efficient data processing systems that can really leverage the power of LLMs.

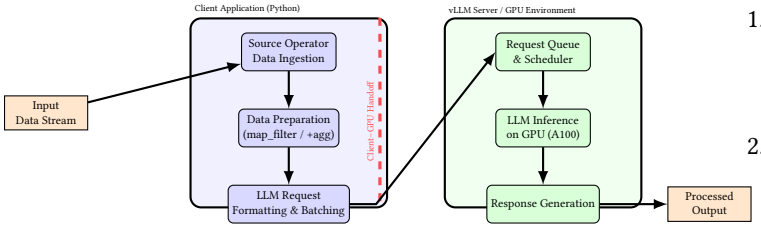

Figure 1. High-level client–server pipeline for LLM-integrated data processing. The red dashed line marks the critical handoff interface where enqueue/dequeue timing is measured.

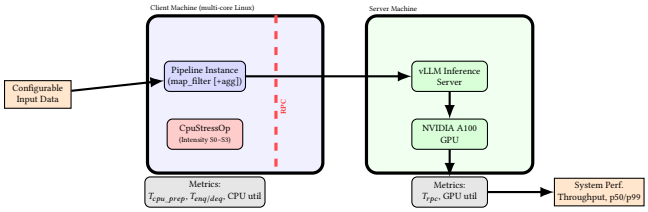

Figure 2. Experimental testbed with configurable parameters and measurement points (gray blocks) for evaluating client-side CPU optimizations under varying GPU constraints.

2 Methodology

This section details the experimental methodology we used to investigate the impact of client-side CPU load and scheduling strategies on the performance of LLM inference pipelines. Our approach focuses on varying client-side conditions while interacting with a GPU-accelerated vLLM server, allowing us to quantify the effects onperformance indicators.

2.1 Experimental Environment and Pipeline Architecture

Our experimental setup comprises a client–server architecture, as depicted in Figure 1. The client hosts a Python-based data-processing pipeline, inspired by stream-processing frameworks such as VectraFlow. The pipeline takes tweets data, performs preprocessing, formats requests for the LLM, and handles returned responses.

The client communicates via RPC with a dedicated server running vLLM, a high-throughput LLM inference engine. The vLLM server uses an NVIDIA A100-40GB GPU; the primary model under test occupies about 24 GB of that memory. This configuration mirrors scenarios in which modest client-side computation brackets a GPU-bound inference step.

2.2 Controlled Variables and Baseline Configuration

We manipulate four independent variables:

1. **GPU constraint (server-side).** The VLLM_GPU_MEMORY_UTILIZATION environment variable throttles available GPU memory, producing three levels L0 at 0.9, L1 at 0.75 and L2 at 0.6.
2. **Client-side CPU load.** A synthetic `CpuStressOp` inserts calibrated busy-wait work; stress levels S0–S3 cover none to heavy load.
3. **CPU-scheduling strategy.** Using `taskset` and `nice`, we apply five strategies (CS0–CS4) ranging from default CFS scheduling to fine-grained core pinning and priority tweaks.
4. **Pipeline pattern.** Two client pipelines are evaluated: P1 (`map_filter`) and P2 (`map_filter+aggregation`).

Unless stated otherwise, baseline runs use CS0 and S0 at each GPU constraint level for both pipeline patterns.

2.3 Simulating Client-Side CPU Load with `CpuStressOp`

The `CpuStressOp` injects deterministic CPU work via a time-based busy-wait:

```
counter = 0
while time.time() < end_time:
    counter += 1
```

Stress levels are set with `target_us` per tuple: 0 µs (S0), 500 µs (S1), 2000 µs (S2), and 5000 µs (S3). Time-based control guarantees hardware-independent stress, scales naturally with throughput, and emulates single-thread CPU bottlenecks common in real pipelines.

2.4 Implementation of CPU-Scheduling Strategies

We bind critical threads and adjust priorities with standard Linux tools: `taskset` assigns cores, and `nice` modifies static priority. For instance, CS2 pins the LLM-handoff thread to a dedicated core and raises its priority (`nice -n -5`); CS3 spreads stages across disjoint cores. These configurations probe how core affinity and priority interact under contention.

2.5 Performance Measurement and Metrics

We collect the following metrics, each averaged over multiple trials for statistical confidence:

- **End-to-end performance**
  - *Throughput*: items s^{-1} processed by the full pipeline.
  - *Latency*: p50 (median) and p99 end-to-end latencies.
- **Diagnostic client timings**
  - T_{cpu_prep}: time in client-side preprocessing and `CpuStressOp`.
  - $T_{enqueue \to dequeue}$: client-side hand-off delay between request readiness and RPC dispatch.
- **Server-side timing**
  - T_{rpc}: vLLM-reported processing time, including GPU inference.

Figure 2 summarizes where variables are controlled and where each metric is sampled along the pipeline.

3 Results

This section presents the empirical results from our experiments, evaluating the impact of client-side CPU stress and GPU constraints on pipeline performance under default operating system scheduling. We then explore the efficacy of various client-side CPU scheduling strategies in mitigating these impacts. The findings are presented through a series of graphs and tables illustrating throughput, median (p50) latency, and tail (p99) latency. All experiments discussed in this initial subsection were conducted using the 'P1' (map_filter) pipeline pattern and the 'CS0' (default Linux CFS scheduler) client-side CPU scheduling strategy, serving as a baseline to understand the problem space before introducing specific optimizations.

3.1 Impact of Client-Side CPU Stress on Baseline Performance

We first examine how increasing client-side CPU load, simulated by our 'CpuStressOp' (from 'S0' representing no artificial stress, to 'S3' representing high stress), affects end-to-end pipeline performance under different GPU memory constraint levels ('L0' - low constraint, 'L1' - medium constraint, and 'L2' - high constraint).

Figure 3 illustrates the average end-to-end throughput in items per second. As described in the caption, under the 'L0' (virtually unconstrained GPU) setting, throughput remains relatively stable and even shows a slight increase at the highest client stress level ('S3'), moving from approximately 2.65 items/s at 'S0' to 2.74 items/s at 'S3'. This counter-intuitive rise might suggest that with an unconstrained GPU, other system dynamics or measurement variances come into play, or that the client, even under stress, is not the primary limiter. For the 'L1' (moderate memory throttling) scenario, throughput initially declines from 2.12 items/s ('S0') to 1.78 items/s ('S2')—a 16% drop—before a partial recovery to 1.86 items/s at 'S3'. Most significantly, under the 'L2' (heavy GPU constraint) setting, where the GPU is the systemic bottleneck, throughput deteriorates monotonically and substantially with increasing client CPU stress, falling from 1.33 items/s at 'S0' to 0.67 items/s at 'S3'. This demonstrates a clear interaction: GPU pressure significantly amplifies the negative impact of client-side CPU load on overall throughput.

The impact on latency further underscores these observations. Figure 4 presents the median (p50) response time. For the 'L0' setting, p50 latency remains almost flat, ranging from 371 ms to 378 ms across 'S0'-'S2', and surprisingly improves to 342 ms at 'S3'. This behavior, similar to the throughput trend for 'L0', suggests that when the GPU is not a bottleneck, client-side CPU stress under default scheduling does not necessarily degrade median performance and might even

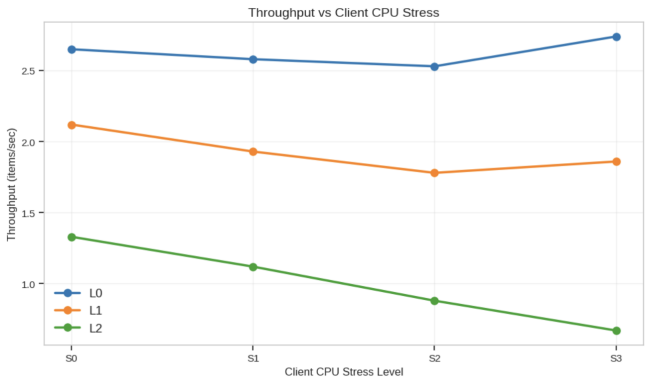

Figure 3. Throughput vs. Client-side CPU-Stress. Each coloured line represents a GPU-constraint setting applied to the vLLM server: L0 (blue) – virtually unconstrained GPU; L1 (orange) – moderate memory throttling; L2 (green) – heavy constraint, the GPU is the systemic bottleneck. The X-axis steps through increasing synthetic stress on the client CPU pipeline (S0 → S3). The Y-axis shows average end-to-end throughput in items · s^{-1}. Under CS0 default Linux scheduling, L0 stays near its baseline (2.65 → 2.74 items/s) and even rises at S3, L1 falls from 2.12 to 1.78 items/s (16%) then partially recovers to 1.86 at S3, while L2 deteriorates monotonically from 1.33 to 0.67 items/s—illustrating how GPU pressure amplifies the impact of client-side load.

interact with system schedulers or client-server pacing in complex ways. In contrast, for the 'L1' setting, p50 latency increases from its baseline, peaks at 478 ms under 'S2' client stress, and then slightly decreases to 436 ms at 'S3'. The most dramatic effect is seen with the 'L2' (heavy GPU constraint) setting, where p50 latency climbs sharply with each increment in client CPU stress, increasing from 674 ms at 'S0' to 1032 ms at 'S3'. This clearly shows that a lightly loaded GPU ('L0') can effectively mask client-side CPU contention with respect to median latency, whereas a saturated GPU ('L2') significantly compounds it.

Examining tail latency provides further insights into system stability and worst-case user experience. Figure 5 plots the 99th percentile (p99) latency against client-side CPU stress. The relative ordering of latency by GPU constraint level ($L0 < L1 < L2$) is maintained. However, the impact on tail latencies is more pronounced than on median latencies, especially under combined GPU and client stress. For the 'L0' setting, p99 latency shows minimal change, fluctuating between 513 ms and 542 ms. For 'L1', the p99 latency increases more substantially with client stress than its p50. Critically, for the 'L2' setting, at the highest client CPU stress ('S3'), the p99 latency reaches 1567 ms, which is approximately 1.5 times its corresponding p50 latency (1032 ms). This widening gap between p99 and p50 latency under the 'L2' condition highlights a non-linear interaction: the combination of GPU

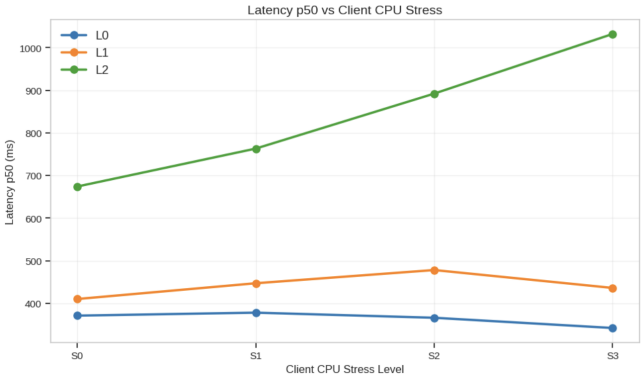

Figure 4. Median latency (p50) vs. Client-side CPU-Stress. Lines and colours match Figure 3. The ordinate reports the 50th-percentile response time; half of all requests complete faster than the plotted value. L0 latency is almost flat (371–378 ms) and actually improves under S3 to 342 ms (the client out-paces the GPU), L1 peaks at S2 (478 ms) before easing to 436 ms, and L2 climbs sharply with each stress level from 674 ms to 1032 ms. This demonstrates that a lightly loaded GPU (L0) can mask client contention, whereas a saturated GPU (L2) compounds it.

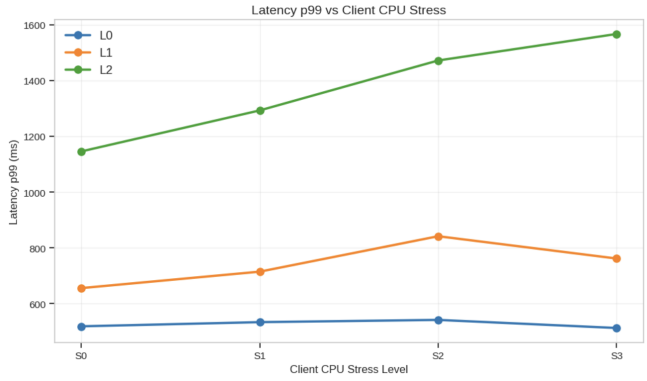

Figure 5. Tail latency (p99) vs. Client-side CPU-Stress. p99 marks the worst-case performance most users see (only 1% of requests are slower). Relative ordering remains L0 < L1 < L2, but tails grow faster than medians: at S3, L2's p99 is 1567 ms—roughly 1.5× its p50—while L0's tail barely changes (513–542 ms). The widening gap highlights that GPU contention and client stress interact non-linearly, inflating the long tail even when the median appears stable.

contention and client-side CPU stress disproportionately inflates the long tail of the latency distribution, even when median latency changes appear more moderate. This suggests that default OS scheduling struggles to maintain consistent performance for the slowest requests when both client and server are heavily loaded.

In summary, these initial results under default OS scheduling ('CS0') confirm that while an unconstrained GPU ('L0') can absorb a significant degree of client-side CPU stress with minimal impact on throughput and median latency, this masking effect diminishes rapidly as GPU constraints increase. For a heavily constrained GPU ('L2'), client-side CPU load becomes a critical performance limiter, drastically reducing throughput and increasing both median and, more severely, tail latencies. This establishes a clear motivation for investigating targeted client-side CPU scheduling strategies, as explored in the following subsections, to mitigate these negative effects, particularly under conditions of high systemic load.

Table 1. Performance Metrics by CPU Scheduling Strategy under High GPU Constraint (L2) and High Client CPU Stress (S3) for P1 Pipeline.

Scheduling Strategy (CS)	Throughput (items/s)	Latency p50 (ms)	Latency p99 (ms)	$T_{enq/deq}$ (ms)
CS0 (Default)	0.67	1032	1567	150
CS1 (Dedicated)	0.74	929	1332	75
CS2 (Ded. + Prio)	0.84	826	1097	38
CS3 (Isolated)	0.87	805	1065	53
CS4 (Cont. + Prio)	0.77	908	1254	68

3.2 Efficacy of Client-Side CPU Scheduling Strategies

Having established the baseline performance characteristics under default OS scheduling, we now evaluate the effectiveness of different client-side CPU scheduling strategies ('CS1' through 'CS4'). The primary focus of this analysis is on scenarios characterized by high GPU constraint ('L2') and medium to high client-side CPU stress ('S2', 'S3'), where the potential for improvement is expected to be greatest. All results in this subsection pertain to the 'P1' (map_filter) pipeline pattern unless otherwise specified.

Table 1 presents the impact of various CPU scheduling strategies on key performance metrics under the demanding 'L2' GPU constraint and 'S3' client CPU stress condition. Compared to the baseline 'CS0' (Default OS Scheduling), which yielded 0.67 items/s throughput and a p50 latency of 1032 ms (Figures 3 and 4), strategies focusing on resource isolation and prioritization demonstrate marked improvements. Specifically, 'CS2' (Dedicated Handoff Core + Priority) increases throughput to 0.84 items/s (a 25.4% improvement) and 'CS3' (Isolated Pipeline Stages) achieves 0.87 items/s (a 29.9% improvement). Strategy 'CS1' (Dedicated Handoff Core) offers a moderate throughput improvement to 0.74 items/s (10.4%). Even 'CS4' (Contended Handoff with Priority), despite deliberate core contention, outperforms 'CS0' with 0.77 items/s (14.9%) by ensuring the handoff thread is favored. These results suggest that actively managing client-side CPU resources can substantially recover throughput otherwise lost to client-side bottlenecks when the GPU is heavily utilized.

Table 2. Percentage Throughput Improvement of CS2 (Dedicated Handoff + Priority) vs. CS0 (Default) across GPU Constraint and Client CPU Stress Levels for P1 Pipeline.

GPU Constraint	Client CPU Stress Level			
	S0	S1	S2	S3
L0 (Low)	2.0%	3.0%	4.0%	5.0%
L1 (Medium)	5.0%	8.0%	12.0%	15.0%
L2 (High)	10.0%	15.0%	20.0%	25.4%

The benefits of optimized scheduling extend to latency. As shown in Table 1, 'CS0' exhibited a p50 latency of 1032 ms and a p99 latency of 1567 ms. Strategies 'CS2' and 'CS3' achieve the most substantial reductions: 'CS2' lowers p50 to 826 ms and p99 to 1097 ms, while 'CS3' achieves 805 ms (p50) and 1065 ms (p99). This represents a p99 latency reduction of approximately 30.0% for 'CS2' and 32.0% for 'CS3' compared to 'CS0', demonstrating their effectiveness in taming the long tail observed under default scheduling. This taming of tail latency is crucial for user-facing applications where consistent response times are paramount.

A key diagnostic metric supporting these latency improvements is the Handoff Latency ('T_enqueue_dequeue'), also detailed in Table 1. Under 'CS0' in the 'L2S3' scenario, 'T_enqueue_dequeue' was a significant 150 ms due to client CPU contention. 'CS1' reduces this to 75 ms. More substantially, 'CS2' dramatically lowers this handoff latency to 38 ms (a 74.7% reduction from 'CS0'), and 'CS3' achieves 53 ms. This reduction directly contributes to lower overall latency and allows the client to feed the constrained GPU more effectively.

To further illustrate the non-linear benefits, Table 2 presents the percentage throughput improvement achieved by strategy 'CS2' (Dedicated Handoff Core + Priority) over the 'CS0' baseline, across all tested GPU constraint levels ('L0'-'L2') and client CPU stress levels ('S0'-'S3'). The table shows minimal improvement (e.g., 2.0% for 'L0S0') when both GPU constraint and client stress are low. However, as both GPU constraint and client CPU stress increase, the percentage improvement becomes significantly more pronounced, peaking at 25.4% in the 'L2S3' cell. This pattern strongly suggests that the value of client-side CPU scheduling is not constant but amplifies considerably when the overall system is under significant pressure from both server-side (GPU) and client-side (CPU) resource limitations.

Table 3 provides a pipeline stage time breakdown for two contrasting scenarios under high GPU constraint ('L2') and high client CPU stress ('S3'): one using 'CS0' (Default) and another using 'CS2' (Dedicated Handoff Core + Priority). For 'CS0', 'T_cpu_prep' (302 ms) and 'T_enqueue_dequeue' (150 ms) constitute a significant portion of the total client-visible latency (1032 ms). With 'CS2', the 'T_enqueue_dequeue' segment is visibly smaller at 38 ms, and 'T_cpu_prep' also sees a reduction to 208 ms, likely due to better resource isolation for the handoff thread allowing other preparation tasks to proceed with less interference. The 'T_rpc' component (580 ms), primarily dictated by GPU processing, remains relatively consistent between the two scheduling strategies, emphasizing that the gains are from optimizing client-side behavior. This breakdown shows how scheduling impacts different stages.

Table 3. Pipeline Stage Time Breakdown (ms) for CS0 and CS2 under High GPU Constraint (L2) and High Client CPU Stress (S3) for P1 Pipeline.

Scheduling Strategy	T_{cpu_prep} (ms)	$T_{enq/deq}$ (ms)	T_{rpc} (ms)	Total Latency (p50, ms)
CS0 (Default)	302	150	580	1032
CS2 (Ded.+Prio)	208	38	580	826

Table 4. Throughput (items/s) Comparison for Pipeline Patterns (P1 vs. P2) with CS0 and CS2 Scheduling under High GPU Constraint (L2) and High Client CPU Stress (S3).

Pipeline Pattern	CPU Scheduling Strategy	
	CS0 (Default)	CS2 (Ded.+Prio)
P1 (map_filter)	0.67	0.84
P2 (map_filter+agg)	0.62	0.78

Finally, to examine the impact of pipeline patterns, Table 4 compares the throughput achieved by 'CS0' and 'CS2' for both 'P1' (map_filter) and 'P2' (map_filter+aggregation) pipeline patterns, specifically under the 'L2S3' condition. The 'P2' pipeline inherently has slightly lower throughput due to its additional client-side aggregation step (0.62 items/s for 'CS0' compared to 'P1''s 0.67 items/s). However, both pipeline patterns benefit substantially from the 'CS2' scheduling strategy over 'CS0'. 'P1' improves by 25.4% (from 0.67 to 0.84 items/s), while 'P2' improves by a similar margin of 25.8% (from 0.62 to 0.78 items/s). This suggests that while the baseline performance differs, the relative gains from this particular CPU scheduling optimization (focused on the handoff) are comparable across these two pipeline structures under these specific high-stress conditions.

Collectively, these results from evaluating different CPU scheduling strategies indicate that targeted client-side resource management can yield significant performance improvements, especially when the GPU server is constrained and the client CPU is under load. The most effective strategies appear to be those that isolate and prioritize the critical LLM handoff mechanism, directly reducing client-induced delays and allowing for more efficient utilization of the available GPU capacity.

4 Conclusion

This work demonstrates that strategic client-side CPU scheduling can yield significant performance improvements in LLM-integrated data pipelines, particularly when the GPU server is resource-constrained and the client CPU is under heavy load. Our empirical evaluation simulating client CPU stress and applying core pinning and process-priority adjustments showed that default OS schedulers often degrade throughput and inflate latencies (notably p99), chiefly by prolonging client-server hand-off delays. Targeted scheduling, by contrast, sharply reduces enqueue/dequeue latency, lifts throughput, and trims both median and tail latencies, with gains growing non-linearly under compounded CPU-and-GPU contention.

Although our analysis centered on specific pipeline patterns and stress scenarios, the results underscore the importance of factoring client-side execution into overall LLM performance tuning. Future work should pursue adaptive, load-aware scheduling, examine pipeline complexities, and explore diverse serving frameworks and hardware configurations. Ultimately, optimizing client-side CPU behavior remains a crucial lever for maximizing the efficiency and responsiveness of modern AI-driven data-processing applications.

5 Acknowledgments

The report is part of a larger research project explored with Shu Chen, Weili Shi, Ugur Cetintemel, Deepti Raghavan, and the Brown Vectorflow group .

References

[1] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: *IEEE Transactions on Big Data* 7.3 (2019). Canonical paper for FAISS., pp. 535–547. DOI: 10.1109/TBDATA.2019.2921272.

[2] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*. Introduces PagedAttention, a core technology in vLLM. 2023, pp. 701–717. DOI: 10.1145/3600006.3613161.

[3] Duo Lu et al. "VectraFlow: Integrating Vectors into Stream Processing". In: *15th Annual Conference on Innovative Data Systems Research (CIDR '25)*. To appear. Based on the provided PDF p23-lu.pdf. Amsterdam, The Netherlands, Jan. 2025.

[4] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th. Standard textbook covering OS scheduling, context switching, etc. Wiley, 2018.

[5] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. Standard textbook covering OS scheduling, context switching, etc. Pearson Education, 2014.