DOI: https://doi.org/10.36910/6775-2524-0560-2026-62-10
UDC 004.382
Fedorovych Illia, Postgraduate Student
https://orcid.org/0009-0003-7739-9689
Osukhivska Halyna, Ph.D., Associate Professor
https://orcid.org/0000-0003-0132-1378
Ternopil Ivan Puluj National Technical University, Ternopil, Ukraine

GPU-ACCELERATED LEMMATIZATION WITH CUDA AND RAPIDS ACCELERATOR FOR SPARK

Fedorovych I., Osukhivska H. GPU-accelerated lemmatization with CUDA and RAPIDS accelerator for Spark. Lemmatization is one of the few preprocessing stages in NLP pipelines that is still predominantly CPU-bound, even as tokenization, vectorization, and model inference increasingly rely on high-throughput GPU execution. Deterministic, dictionary-based morphological analyzers deliver stable and linguistically correct results, but their sequential nature, complex branch divergence, and interoperability overhead significantly limit processing throughput in large-scale distributed systems. In this work, we propose a GPU-native deterministic lemmatizer for Ukrainian, implemented as a custom CUDA kernel and integrated into Apache Spark via a RAPIDS-accelerated JNI UDF. A large morphological dictionary (VESUM/dict_uk) is compiled into a compact trie layout optimized for coalesced memory traversal and fully parallel token processing, preserving accuracy while eliminating CPU constraints and reducing end-to-end latency. We evaluate three Spark pipelines — Python with pymorphy3, Java with Morfologik, and our CUDA implementation — on corpora up to 175.6M tokens. The GPU approach achieves 35.77M tokens/s, outperforming the optimized Java baseline by 8.8× and the Python baseline by up to 120×. Our results demonstrate that classical morphological analysis — historically treated as inherently sequential — can be re-architected into an efficient, scalable, and bandwidth-bound GPU operation suitable for real-world big-data workflows.
Keywords: GPU acceleration, CUDA, Apache Spark, RAPIDS Accelerator, lemmatization, morphological analysis, finite-state methods, big data processing, parallel text processing.

Федорович І.А., Осухівська Г.М. Лематизація з прискоренням GPU за допомогою CUDA та RAPIDS-акселератора для Spark. Лематизація є одним з небагатьох етапів попередньої обробки в конвеєрах NLP, який все ще переважно залежить від процесора, навіть попри те, що токенізація, векторизація та виведення моделей все більше залежать від високопродуктивного виконання на GPU. Детерміновані морфологічні аналізатори на основі словників забезпечують стабільні та лінгвістично коректні результати, але їхня послідовна структура, складна логіка розгалужень та витрати на міжплатформну взаємодію суттєво обмежують продуктивність обробки у великомасштабних розподілених системах. У цій роботі запропоновано детермінований лематизатор для української мови, розроблений на GPU як користувацьке ядро CUDA та інтегрований в Apache Spark через JNI UDF з прискоренням RAPIDS. Великий морфологічний словник (VESUM/dict_uk) компілюється в компактний макет trie, оптимізований для об'єднаного проходження пам'яті та повністю паралельної обробки токенів, зберігаючи точність, усуваючи обмеження процесора та зменшуючи наскрізну затримку. Здійснюється оцінка трьох конвеєрів Spark — Python з pymorphy3, Java з Morfologik та запропонованої реалізації CUDA — на корпусах до 175,6 млн токенів. Підхід на GPU досягає 35,77 млн токенів/с, перевершуючи оптимізований базовий рівень Java на 8,8× та базовий рівень Python до 120×. Отримані результати демонструють, що класичний морфологічний аналіз, який історично розглядався як послідовний за своєю суттю, може бути перепроектований в ефективну, масштабовану та обмежену пропускною здатністю операцію на GPU, придатну для реальних робочих процесів з великими даними.
Ключові слова: прискорення на GPU, CUDA, Apache Spark, прискорювач RAPIDS, лематизація, морфологічний аналіз, методи скінченних станів, обробка великих даних, паралельна обробка тексту.

Scientific problem statement. The field of Natural Language Processing (NLP) is currently experiencing rapid growth, driven by the increasing need to analyze massive textual datasets in real time. As applications scale, the efficiency of fundamental preprocessing steps becomes more important. Among these, lemmatization remains a key component for normalizing text data. Lemmatization is one of the fundamental preprocessing tasks in natural language processing (NLP). Lemmatization is the process of reducing a word form to its canonical base form, or lemma. The choice of lemma is determined by linguistic convention and may differ by part of speech and language: for example, in Ukrainian, nouns are conventionally lemmatized to the singular nominative form, whereas verbs use the infinitive. In morphology-rich languages, lemmatization is essential for preserving grammatical distinctions that would otherwise be lost with simpler normalization methods. Unlike stemming, it handles inflectional variation, morphophonological alternations, homonymy, and irregular forms, thus providing linguistically meaningful normalization [1], [2]. When combined with part-of-speech tagging and full morphological analysis, lemmatization underpins downstream tasks such as syntactic parsing, sentiment analysis, named-entity recognition (NER), and spell checking. Reducing vocabulary sparsity while retaining grammatical information improves both the accuracy and performance of NLP systems in applications such as search indexing, information retrieval, and training of large language models. While modern NLP pipelines increasingly rely on GPU acceleration, text preprocessing is still dominated by CPU-bound tools or, in GPU-enabled cases, probabilistic machine-learning models [3], [4]. Deterministic CPU preprocessing often lags behind GPU inference, creating a bottleneck that slows down end-to-end pipelines [5]. As downstream tasks such as semantic indexing and similarity measurement increasingly utilize hardware acceleration, the latency of the initial lemmatization stage becomes a primary bottleneck [6]. Furthermore, while extensive benchmarks exist for AI model training and inference on accelerators, the performance characteristics of morphological preprocessing on these devices remain underexplored [7]. Ensuring strict determinism on massively parallel SIMT architectures is a major challenge and often requires specific hardware or software synchronization mechanisms [8], making this imbalance even more pronounced in real-time and big-data environments.

The relevance of this study is driven by three converging factors: the morphological complexity of the Ukrainian language, which requires computationally intensive analysis; the exponential growth of big data volumes; and the critical demand for high-speed processing in modern production environments.

Although GPU acceleration has transformed many areas of NLP, including neural inference and string manipulation (e.g., tokenization and regex), there has been no deterministic GPU-native morphological analyzer. Existing lemmatizers for morphologically rich languages such as Ukrainian remain CPU-based, making them unsuitable for large-scale or real-time GPU-centric pipelines.

In Ukrainian NLP, the most comprehensive deterministic resource is VESUM (*Velykyi Elektronnyi Slovnyk Ukrainskoi Movy*, Large Electronic Dictionary of Ukrainian), a large morphological dictionary designed for both analysis and synthesis, containing over 400 000 lemmas with rich morphological features and paradigm information [9]. VESUM's scope extends beyond standard vocabulary to include proper names, abbreviations, dialectal and archaic words, and non-standard or deprecated forms. It also features a dynamic tagging component that interacts with the GRAC corpus through an iterative feedback loop: new or unrecognized tokens surfaced in corpus processing are reviewed by linguists and integrated into the lexicon, keeping coverage synchronized with real-world usage. VESUM has been successfully integrated into applications such as LanguageTool and Lucene, demonstrating its practical utility in text correction and search. Complementary to dictionary-driven resources, Babych [10] introduced a knowledge-light unsupervised paradigm induction method to expand lexicons for named entities and neologisms—categories often absent from static morphological databases. By combining small inflection tables with unannotated corpora, Babych's method enables rapid extension of morphological coverage without retraining or manual annotation. Together, VESUM's dynamic lexicon and Babych's paradigm induction represent a deterministic-first framework capable of adapting to lexical change (e.g., decommunization toponyms, new feminine forms, orthographic reforms).

In this work, we propose a hybrid architecture that bridges the gap between CPU-bound morphology and GPU-accelerated pipelines. This work is dedicated to solving the latency delays in distributed text processing by re-architecting the lemmatization process for massive parallel execution.

Research analysis. Existing lemmatizers are typically embedded within broader morphological analyzers and fall into two categories: rule-based and machine-learning-based. Rule-based systems such as pymorphy3 (Python) and Morfologik (Java) rely on finite-state automata compiled from morphological dictionaries. Deterministic CPU baselines typically rely on highly optimized Finite State Automata (FSA) construction, such as those described by [11] and reimplemented for efficiency by [12]. Prior work [13] has successfully demonstrated that dynamic programming and string alignment algorithms can be accelerated on GPUs using SIMT abstractions; however, morphological dictionary lookups have largely remained on CPU. Despite the rise of neural methods, rule-based and dictionary-based analyzers remain essential for high-precision tasks in morphologically rich languages, as seen in a recent paper on Latin and Slavic languages [14]. In contrast, machine-learning-based systems—such as Stanza and NLP-Cube—predict lemmas probabilistically using neural architectures. These models can leverage GPU acceleration for training and inference but lack deterministic guarantees, and their accuracy strongly depends on the quality and coverage of training data. For South Slavic languages, CLASSLA-Stanza extends the neural paradigm with language-specific optimizations, yet it remains probabilistic and non-deterministic [4], [15]. Thus, while GPUs are widely used for neural inference, deterministic morphology itself remains CPU-bound.

Rule-based morphological analyzers for other languages (e.g., Spanish, derived from spell-checking lexicons, and Kazakh, with explicit morphophonological rules) demonstrate the portability of deterministic methods to other morphology-rich languages [16], [17]. These systems are grounded in the finite-state paradigm for morphology, based on finite-state automata (FSA) and finite-state transducers (FST), which have long been recognized for their deterministic efficiency in large-scale text processing [1],

[2]. However, these classical CPU-oriented architectures have not been adapted for GPU execution, despite the conceptual similarity between parallel state traversal and GPU thread execution.

Meanwhile, the surrounding infrastructure for large-scale NLP has increasingly adopted GPU acceleration. Prior work has demonstrated that GPU-accelerated Spark and MapReduce pipelines can achieve substantial end-to-end speedups—up to 25× compared to CPU-only Spark—despite the overheads of data transfer and JVM–GPU conversion, highlighting the performance benefits of leveraging GPUs for data-intensive workloads [18]. At the distributed level, Apache Spark remains the industry standard for scalable text analytics, with extensions such as Spark NLP integrating linguistic components into data-intensive workflows. Yet, morphological analysis still typically runs on CPUs, introducing a major latency bottleneck in GPU-centric pipelines. Production reports highlight Spark's impressive scalability but also its persistent overheads from serialization, task scheduling, and I/O. Surveys of the Spark ecosystem further document these performance challenges and emphasize the need for hybrid CPU–GPU optimization in text-processing workloads [5].

From a systems perspective, the computational limits of parallel acceleration are well established. Classic GPU/CPU studies show that many data-parallel algorithms—such as sorting—are ultimately bandwidth-bound rather than compute-bound, a pattern mirrored in lemmatization's dictionary lookup workload [19]. Even when the parallel fraction of a pipeline is aggressively optimized, Amdahl's law predicts diminishing returns once sequential components such as I/O and scheduling dominate runtime. Empirical work in big-data analytics confirms that end-to-end performance improvements plateau as non-parallelizable overheads accumulate [20]. Recent analyses further note that non-GEMM workloads (e.g., string manipulation and irregular computation) behave very differently from dense matrix operations, stressing GPU memory and synchronization subsystems in ways that limit attainable speedups [21]. These observations define a performance horizon for GPU-accelerated NLP. While significant acceleration is possible, it is ultimately constrained by memory bandwidth and system-level orchestration. Furthermore, the end-to-end efficiency of such pipelines is influenced by the choice of Data Lakehouse architecture used for storing high-throughput outputs. Recent comparative research [22] indicates that while formats like Delta Lake optimize for data loading speed, Apache Iceberg provides superior stability and disk space optimization for structured and semi-structured datasets, highlighting the necessity of aligning GPU-native processing with efficient storage layers.

In summary, existing literature highlights three complementary realities. First, deterministic finite-state analyzers deliver linguistically precise morphology but remain sequential and CPU-bound. Second, neural lemmatizers exploit GPUs effectively yet sacrifice determinism and reproducibility. And finally, distributed frameworks such as Spark and RAPIDS provide the hardware and software foundations for large-scale GPU pipelines, but morphological preprocessing persists as a CPU bottleneck. To our knowledge, there is no prior research describing a GPU-native deterministic morphological analyzer or its integration with Spark RAPIDS. Bridging this gap—by designing a parallel CUDA kernel for deterministic lemmatization and embedding it into a GPU-accelerated distributed environment—is the central contribution and motivation of this work.

While GPUs are standard for neural inference, to the best of our knowledge, no prior studies have implemented a fully deterministic GPU-native morphological analyzer for Slavic languages. This paper proposes the first such approach.

Definition of the research goal. The objective of this research is to accelerate morphological processing (lemmatization) through the utilization of GPU kernels and massive parallelism. To achieve this goal, it is proposed to design a parallel algorithm, implement it utilizing CUDA technology, and integrate it with the Apache Spark ecosystem to combine distributed cluster computing with device-level SIMD parallelism.

Presentation of the main material and the justification of the results. Our research is guided by the hypothesis that word-level lemmatization is trivially parallelizable and therefore maps naturally onto the SIMD execution model of modern GPUs. Each token can be processed independently, without inter-token dependencies, making the task well suited for parallel dictionary lookups. For baseline comparison, we implemented three Spark pipelines with different lemmatization backends. The first was a Python UDF pipeline using pymorphy3 as the lemmatizer engine, executed on CPU. The second was a Java UDF pipeline based on Morfologik/Jmorphy; since different Java engines produced statistically negligible differences, we report them jointly as a single baseline. The third pipeline was a CUDA UDF with a custom C++/CUDA lemmatizer. All pipelines were launched via spark-submit from Python to ensure identical execution and result collection procedures.

For baseline comparison, we implemented three Spark pipelines that differ only in the lemmatization backend, allowing us to isolate system-level effects from algorithmic ones. All pipelines shared identical Spark configurations, corpus partitions, and data-collection procedures, and were executed through a unified spark-submit interface from Python. A comparison of the architectural approaches is illustrated in Figures 1 and 2. Figure 1 contrasts the traversal logic of the CPU-based Finite State Automaton (FSA) against the GPU-optimized trie structure. Figure 2 demonstrates the difference in data flow between the traditional CPU-bound pipelines and the proposed parallel GPU execution model.

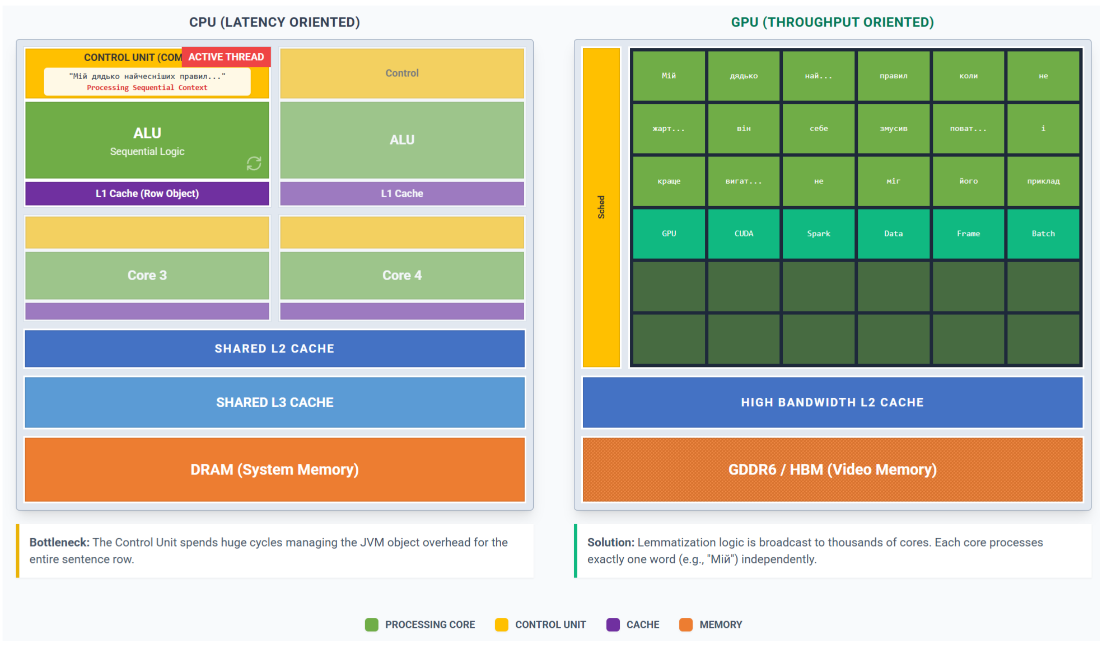
CPU (LATENCY ORIENTED)
CONTROL UNIT (COM
ACTIVE THREAD
"Мій дядько найчесніших правил..."
Processing Sequential Context
ALU
Sequential Logic
L1 Cache (Row Object)
Control
ALU
L1 Cache
Core 3
Core 4
SHARED L2 CACHE
SHARED L3 CACHE
DRAM (System Memory)
GPU (THROUGHPUT ORIENTED)
Sched
Мій
дядько
най...
правил
коли
не
жарт...
він
себе
змусив
поват...
і
краще
вигат...
не
міг
його
приклад
GPU
CUDA
Spark
Data
Frame
Batch
HIGH BANDWIDTH L2 CACHE
GDDR6 / HBM (Video Memory)
Bottleneck: The Control Unit spends huge cycles managing the JVM object overhead for the entire sentence row.
Solution: Lemmatization logic is broadcast to thousands of cores. Each core processes exactly one word (e.g., "Мій") independently.
PROCESSING CORE
CONTROL UNIT
CACHE
MEMORY

Fig. 1. Comparison of execution models: sequential finite-state automaton traversal on CPU (left) versus massively parallel trie traversal on GPU (right).

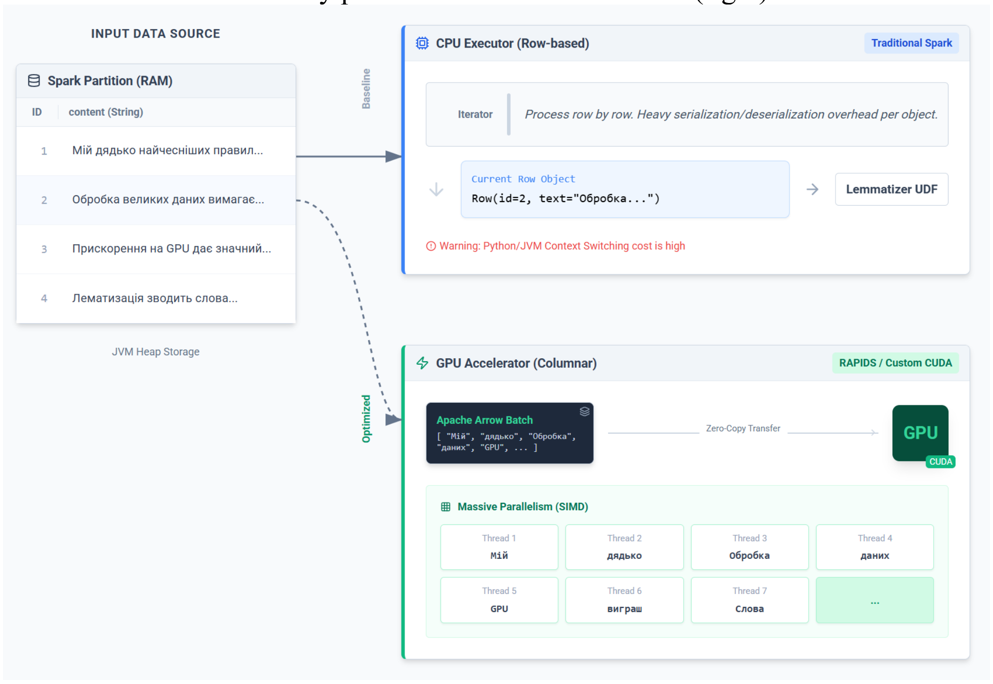
INPUT DATA SOURCE
Spark Partition (RAM)
ID
content (String)
1
Мій дядько найчесніших правил...
2
Обробка великих даних вимагає...
3
Прискорення на GPU дає значний...
4
Лематизація зводить слова...
JVM Heap Storage
Baseline
Optimized
CPU Executor (Row-based)
Traditional Spark
Iterator
Process row by row. Heavy serialization/deserialization overhead per object.
Current Row Object
Row(id=2, text="Обробка...")
Lemmatizer UDF
Warning: Python/JVM Context Switching cost is high
GPU Accelerator (Columnar)
RAPIDS / Custom CUDA
Apache Arrow Batch
["Мій", "дядько", "Обробка", "даних", "GPU", ...]
Zero-Copy Transfer
GPU
CUDA
Massive Parallelism (SIMD)
Thread 1
Мій
Thread 2
дядько
Thread 3
Обробка
Thread 4
даних
Thread 5
GPU
Thread 6
виграш
Thread 7
Слова
...

Fig. 2. Data processing flow: traditional Spark processes data as row objects (baseline), whereas the proposed approach utilizes Apache Arrow for zero-copy columnar batch transfer to the GPU (optimized).

Python UDF is a baseline that represents a typical Python-based Spark pipeline used in data-science workflows. Each token is passed from the JVM (where Spark executes) to a separate Python process via Py4J and Arrow serialization. The UDF executes the pymorphy3 lemmatizer sequentially inside the Python interpreter, returning results row-by-row to Spark. While this design is simple and flexible, it incurs high per-row overhead from cross-language serialization and Python's Global Interpreter Lock (GIL). Consequently, even though Spark itself is distributed, the Python UDF portion executes sequentially within each executor process.

To eliminate cross-language serialization overhead, the second pipeline implements the lemmatizer as a native JVM UDF written in Java. Since Spark executors already run in the JVM, function calls occur in-process with zero marshaling cost. The Morfologik and Jmorphy libraries provide deterministic dictionary-based lemmatization similar to pymorphy3 but implemented in Java. Parallelism is achieved through Spark's task scheduling across partitions, yet within each executor the UDF operates in a multi-threaded CPU context, limited by the number of physical cores and by CPU memory bandwidth. This setup removes the Python overhead but remains constrained by the sequential lookup logic of the finite-state automaton.

The third pipeline integrates a GPU-native lemmatizer implemented in C++/CUDA. Spark executors invoke it through a Java Native Interface (JNI) wrapper that links the JVM to compiled shared libraries. The JNI layer transfers token batches to the CUDA runtime, which stores the precompiled trie-based morphological dictionary in device memory (arrays of states, transitions, and lemma indices). The kernel processes thousands of tokens in parallel, performing branchless trie traversal optimized for coalesced memory access. Although each call incurs host–device data-transfer overhead, the massive SIMD parallelism of the GPU compensates for it by executing hundreds of thousands of lookups concurrently. Results are copied back to host memory and returned through JNI to Spark, then to the RAPIDS accelerator stack for further processing. This architecture uniquely combines two levels of parallelism: Spark manages data distribution across cluster nodes (macro-parallelism), while the GPU executes massive SIMD operations on token batches within each node (micro-parallelism).

Side-by-side comparison of three Spark pipelines is presented in Figure 3.

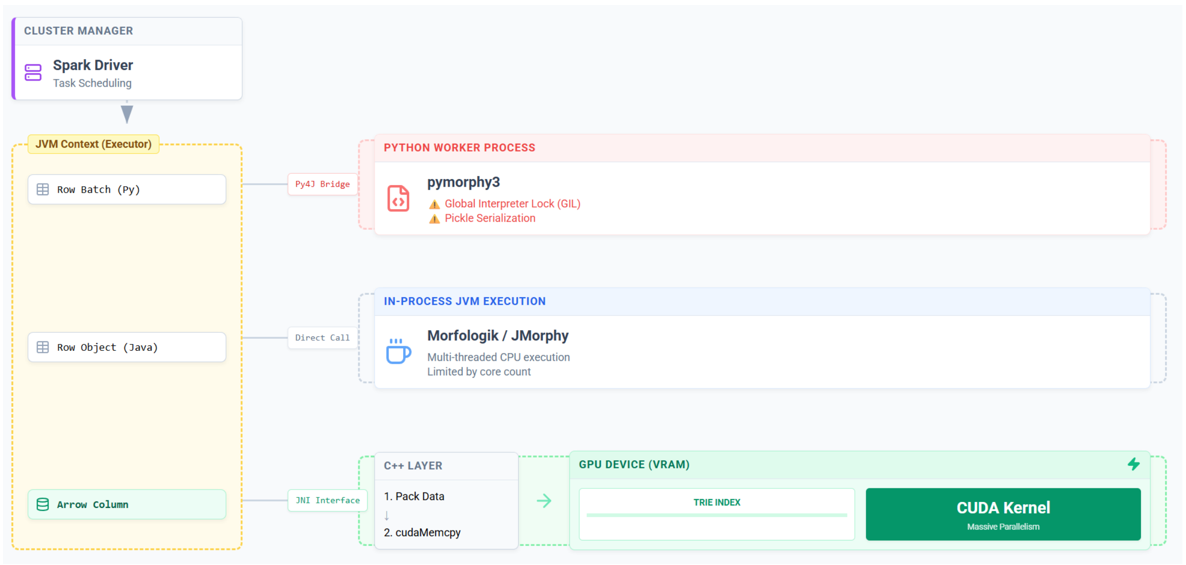

Fig. 3. Architectural comparison of the three experimental pipelines. The Python UDF suffers from serialization and GIL overhead; the Java UDF is limited by CPU core count; the CUDA UDF leverages JNI and massive GPU parallelism.

The three backends illustrate distinct execution models, which are presented in Table 1.

Table 1. Spark pipeline execution models

Pipeline	Language Boundary	Parallelism	Main Bottleneck
Python UDF	JVM → Python (Py4J/Arrow)	None (per-row)	Serialization + GIL overhead
Java UDF	In-JVM	Task-level CPU threads	Memory bandwidth + sequential FSA traversal
CUDA UDF	JVM → JNI → CUDA	Massive SIMD GPU	Host–device transfer + Spark scheduling

Conceptually, Spark's distributed tasks feed token batches to each backend: the Python UDF spends most time in communication; the Java UDF uses CPU threads efficiently but scales only up to core count; and the CUDA UDF, despite extra JNI and PCIe transfers, leverages thousands of concurrent GPU threads to achieve an order-of-magnitude higher throughput. This architectural hierarchy explains the measured performance differentials. In accordance with Amdahl's law, once lemmatization becomes GPU-parallelized, overall pipeline latency is dominated by non-parallelizable Spark stages (I/O, scheduling, and serialization), setting a practical ceiling on further acceleration.

The dictionary was built from the [dict_uk] morphological lexicon, compiled into a trie structure for efficient wordform lookup. To mitigate the thread divergence and irregular memory access patterns common in graph-like traversals on GPUs, the dictionary was compiled into a compact trie layout optimized for coalesced memory traversal [23]. We utilize the CUDA ecosystem, which remains the de facto standard for high-performance AI model serving due to its mature software stack [24], to implement the lemmatization kernel. The CUDA kernel performs parallel trie traversal, with states and transitions stored in device memory arrays optimized for coalesced access. Additionally, the RAPIDS framework provides GPU-optimized dataframes (cuDF) and SQL layers, enabling end-to-end data science workflows entirely within GPU memory [18]. Lemmas are retrieved through direct indexing into a device-resident lemma array. Integration with Spark was achieved through a user-defined function (UDF) that wraps the CUDA kernel and exposes it via the RAPIDS accelerator.

Experiments were conducted on a system with an AMD Ryzen 7 7800X3D CPU and an NVIDIA GeForce RTX 4080 SUPER GPU. Rather than reporting raw FLOP counts, we evaluate performance as a function of input size (number of tokens). This approach reflects the fact that lemmatization is a memory-bound task, where dataset scale and bandwidth utilization are the primary determinants of performance. Two datasets were used: a small corpus of ~4,000 Ukrainian news articles (807,607 tokens), and a large corpus of ~924,000 articles (175,583,276 tokens). The combined vocabulary contained ~6,000,000 unique wordforms.

Performance was measured using multiple metrics: throughput (tokens per second), wall-clock runtime, and latency distributions (p50/p95/p99). Each experiment was repeated n=5 times, and results are reported with 95% confidence intervals.

The results were obtained on Spark 4.0, Java 21, Python 3.12, C++20, RAPIDS 25.08, and CUDA 12.8. To ensure results stability, each pipeline was run 5 times. The results are presented in Table 2.

Table 2. Spark UDF pipeline results

Corpus size	Engine	Tokens	E2E mean (s)	95% CI (s)	Throughput (tok/s)
Small	Python	807 607	18.98	±1.38	42 548
	Java	807 607	1.37	±1.27	590 084
	CUDA	807 607	0.48	±0.57	1 675 327
Large	Python	175 583 276	592.99	±2.77	296 099
	Java	175 583 276	43.27	±1.95	4 058 064
	CUDA	175 583 276	4.91	±1.37	35 769 243

On the small corpus (807K tokens), the Python baseline required 18.98 s (42.5K tok/s). The Java baseline improved performance to 1.37 s (590K tok/s). The CUDA implementation further reduced runtime to 0.48 s (1.68M tok/s), yielding a 39.4× speedup vs Python and 2.8× vs Java.

On the large corpus (175.6M tokens), Python took 593 s (296K tok/s), Java 43.3 s (4.06M tok/s), and CUDA only 4.91 s (35.8M tok/s). This corresponds to a 120.8× speedup over Python and 8.8× over Java. This massive disparity is consistent with broader observations that programming language choice significantly impacts data processing efficiency in Apache Spark, where Scala and Java often outperform Python in large-scale, complex ETL operations [25].

Results of end-to-end runtime for each pipeline and corpus are shown in Figure 4. Throughput metrics are presented in Figure 5. Table 3 shows the relative speedups of one pipeline over the other given the same corpus size.

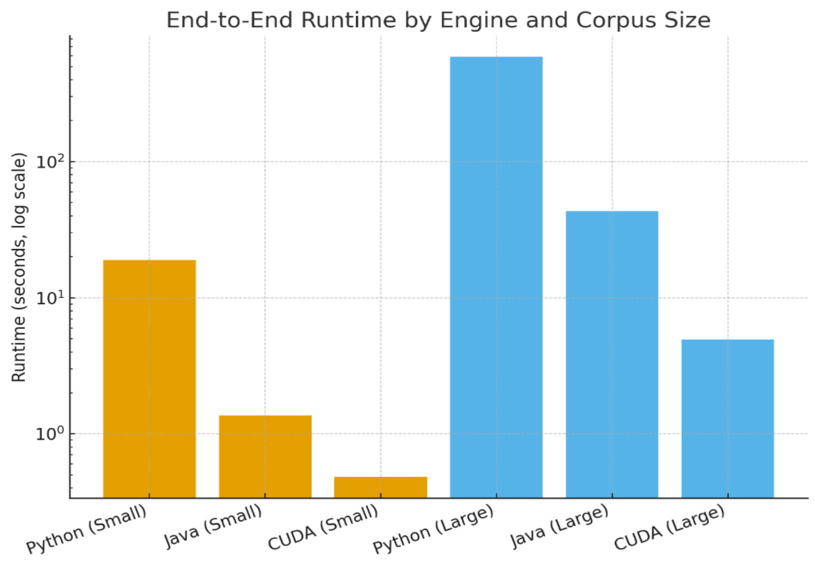

Fig. 4. Comparison of pipeline runtimes in seconds (lower is better)

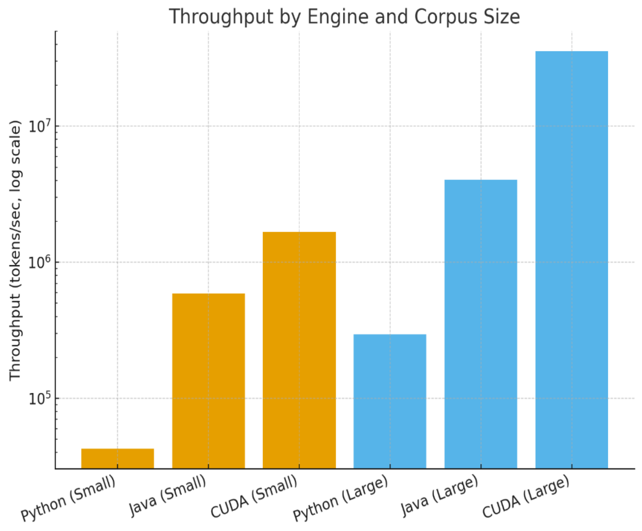

Fig. 5. Comparison of pipeline throughputs in tokens per second (higher is better)

Table 3. Runtime speedup comparison

Corpus size	Java vs Python	CUDA vs Python	CUDA vs Java
Small (0.8M tokens)	13.9×	39.4×	2.84×
Large (175M tokens)	13.7×	120.8×	8.81×

The experimental results demonstrate a clear hierarchy of performance across the three pipelines. Python UDFs (pymorphy3) exhibited the slowest performance, with end-to-end throughput below 0.3M tokens/s on the large dataset. This performance reflects the double penalty of (a) high interpreter overhead in Python UDF execution and (b) lack of efficient vectorization in the underlying morphological analyzer.

By contrast, the Java UDF implementation (Morfologik/Jmorphy) achieved ~4M tokens/s on the same dataset, corresponding to a ~13× speedup relative to Python. These results confirm earlier reports that JVM-based UDFs substantially reduce serialization/deserialization overhead and benefit from JIT compilation when integrated with Spark. Nevertheless, Java UDFs remain constrained by CPU memory bandwidth and the limited parallelism afforded by 8 physical cores on the experimental setup.

The custom CUDA UDF achieved the highest throughput at ~35M tokens/s, a 120× speedup over Python UDFs and a 9× speedup over Java UDFs. This is consistent with the hypothesis that word-level lemmatization is trivially parallelizable and maps well to the GPU's SIMD execution model. Despite the kernel being relatively naïve (e.g., limited coalescing and no use of shared memory optimizations), the GPU's massively parallel execution compensated for inefficiencies, and throughput scaled nearly linearly with input size until Spark I/O stages dominated.

It is notable that even though GPU accumulators indicated significant semaphore wait times and non-negligible PCIe transfer overhead, these did not critically affect end-to-end performance. In effect, the GPU was still underutilized, suggesting further optimizations remain possible. In particular, partition sizing, memory coalescing strategies, and overlapping host-to-device transfers with computation are likely to further reduce bottlenecks.

These findings illustrate two broader points. First, while CPU-based optimization (Python → Java) yields order-of-magnitude improvements, GPU acceleration provides a step change in achievable throughput, with morphological analysis now operating comfortably within single-digit seconds even on corpora exceeding 175M tokens. Second, this case underscores an important systems trade-off: CPUs excel at handling many small Spark partitions, while GPUs prefer fewer, larger batches that maximize intra-device parallelism. Balancing these execution models remains an open challenge for Spark GPU pipelines, but the gains observed here suggest that even without fine-grained optimization, GPU acceleration sets a new baseline for real-time lemmatization in text-streaming workloads.

In line with Amdahl's law, the acceleration we observe is bounded not only by the GPU kernel speed but also by the proportion of the pipeline that remains CPU-bound (e.g., Spark scheduling, I/O, serialization). Even though the CUDA kernel executes lemmatization orders of magnitude faster than CPU baselines, overall speedup plateaus once these non-parallelizable components dominate the runtime. For example, on the large corpus, the raw kernel time averaged ~0.3 s per run, but the measured end-to-end runtime remained ~5–7 s, highlighting that only part of the workload is truly GPU-parallelizable. This explains why speedups saturate around 100× rather than approaching the theoretical parallel factor implied by thousands of GPU cores.

Conclusions and prospects for further research. We presented the first GPU-native deterministic lemmatizer implemented as a custom CUDA kernel using a compact trie layout, integrated into Apache Spark via JNI. The experimental results demonstrate that deterministic lemmatization is highly amenable to GPU acceleration, achieving speedups of up to 120× over Python baselines. Across datasets ranging from 0.8M to 175M tokens, throughput scaled nearly linearly with corpus size until the vocabulary exceeded a few million unique word forms, at which point performance began to plateau. This behavior reflects a scaling law driven by the balance between compute parallelism and memory bandwidth: as dictionary lookups saturate the GPU's memory subsystem, further token-level parallelization yields diminishing returns. This trend is consistent with other memory-bound workloads such as GPU-based sorting or hashing, and confirms that lemmatization primarily depends on efficient random-access traversal rather than arithmetic intensity. This suggests that once optimized for memory coalescing, the proposed kernel design should transfer predictably to other consumer or data-center GPUs.

From a practical standpoint, GPU acceleration becomes beneficial once the input corpus and vocabulary reach certain thresholds. For small datasets (below ~10^5–10^6 tokens), kernel-launch and data-transfer overheads dominate, making CPU execution competitive. Beyond that scale—particularly for corpora above 10^7 tokens and vocabularies under 10^7 unique word forms—the GPU implementation consistently outperforms CPU baselines by one to two orders of magnitude. These thresholds indicate that GPU-native lemmatization is most advantageous in large-scale, high-throughput, or real-time pipelines where preprocessing latency directly impacts end-to-end system responsiveness. In scenarios demanding ultra-low latency, such as real-time text data stream processing, traditional micro-batching can introduce delays that are better mitigated through continuous processing modes [26].

Several limitations frame these findings. First, experiments were conducted on fixed hardware—a single high-end consumer GPU and CPU pair—which limits the generalizability of absolute speedup factors. Second, evaluation was restricted to Ukrainian and the dict_uk / VESUM lexicon, which, although representative of morphology-rich Slavic languages, may not capture edge cases found in other inflectional systems. Third, integration was designed for Apache Spark with RAPIDS Accelerator, so some observed overheads may be specific to Spark's scheduling and I/O model rather than fundamental to the GPU algorithm itself. Finally, our focus on deterministic dictionary lookup excludes hybrid approaches that combine rule-based and neural inference, which could further improve coverage or robustness.

Future work will extend this research in two primary directions. First, we plan to enhance the architectural portability of the implementation by porting the trie-traversal logic to Apple Metal, enabling cross-platform benchmarking between NVIDIA and Apple Silicon architectures. This will be paired with an investigation into native cuDF integration to achieve a fully GPU-resident preprocessing pipeline. Second, we aim to evaluate the role of this kernel as a deterministic grounding mechanism for local LLMs. We will focus on quantifying its impact as a real-time validation layer, specifically testing the hypothesis that high-speed morphological grounding can mitigate hallucinations in small-scale models deployed on edge devices.

Beyond raw throughput, the work highlights an important systems implication: deterministic morphological preprocessing no longer needs to be the CPU bottleneck in GPU-centric NLP and AI workflows. Furthermore, by replacing heavy neural inference with optimized dictionary lookups, this approach addresses the growing concern of energy consumption in large-scale NLP pipelines [27]. Integrating such GPU-resident linguistic operations closes a longstanding gap between accelerated model inference and CPU-bound text preparation, bringing full-stack GPU processing one step closer to reality.

References

1. Karttunen L. Applications of finite-state transducers in natural language processing. *Implementation and Application of Automata*. Berlin, Heidelberg, 2001. P. 34–46. (Lecture Notes in Computer Science; vol. 2088). URL: https://doi.org/10.1007/3-540-44674-5_2.

2. Korzycki M. Finite-state methodology in natural language processing. *Computer Science*. Kraków, 2001. Vol. 3. P. 151–162. URL: https://doi.org/10.7494/csci.2001.3.1.3594.

3. Baxi J., Bhatt B. Recent advancements in computational morphology: a comprehensive survey. URL: https://doi.org/10.48550/arXiv.2406.05424.

4. Boros T., Dumitrescu S. D., Burtica R. NLP-Cube: end-to-end raw text processing with neural networks. *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. Brussels, Belgium, 2018. P. 171–179. URL: https://doi.org/10.18653/v1/K18-2017.

5. A survey on Spark ecosystem for big data processing / S. Tang et al. *IEEE Transactions on Knowledge and Data Engineering*. 2020. P. 71–91. URL: https://doi.org/10.1109/TKDE.2020.2975652.

6. Hardware acceleration for similarity measurement in natural language processing / P. Tandon et al. *International Symposium on Low Power Electronics and Design (ISLPED)*. Beijing, China, 2013. P. 409–414. URL: https://doi.org/10.1109/ISLPED.2013.6629333.

7. Benchmarking the performance and energy efficiency of AI accelerators for AI training / Y. Wang et al. *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020. P. 744–751. URL: https://doi.org/10.1109/CCGrid49817.2020.00-15.

8. GPUDet: A deterministic GPU architecture / H. Jooybar et al. *SIGPLAN Not.* 2013. Vol. 48, no. 4. P. 1–12. URL: https://doi.org/10.1145/2499368.2451118.

9. Starko V., Rysin A. Creating a POS gold standard corpus of modern Ukrainian. *Proceedings of the Second Ukrainian Natural Language Processing Workshop (UNLP)*. Dubrovnik, Croatia, 2023. P. 91–95. URL: https://doi.org/10.18653/v1/2023.unlp-1.11.

10. Babych B. Unsupervised induction of Ukrainian morphological paradigms for the new lexicon: extending coverage for named entities and neologisms using inflection tables and unannotated corpora. *Proceedings of the 7th Workshop on Balto-Slavic Natural Language Processing*. Florence, Italy, 2019. P. 1–11. URL: https://doi.org/10.18653/v1/W19-3701.

11. Incremental construction of minimal acyclic finite-state automata / J. Daciuk et al. *Computational Linguistics*. 2000. Vol. 26, no. 1. P. 3–16. URL: https://doi.org/10.1162/089120100561601.

12. Daciuk J., Weiss D. Smaller representation of finite state automata. *Theoretical Computer Science*. 2012. Vol. 450. P. 10–21. URL: https://doi.org/10.1016/j.tcs.2012.04.023.

13. Liu Y., Schmidt B., Maskell D. L. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*. 2010. Vol. 3, no. 1. P. 93. URL: https://doi.org/10.1186/1756-0500-3-93.

14. Celano G. G. A. Lemmatization and morphological analysis for the Latin dependency treebank. *Studi e Saggi Linguistici*. 2020. Vol. 58, no. 1. P. 21–38. URL: https://doi.org/10.4454/ssl.v58i1.274.

15. Ljubešić N., Terčon L., Dobrovoljc K. CLASSLA-Stanza: the next step for linguistic processing of South Slavic languages. *Conference on Language Technologies and Digital Humanities*. 2024. URL: https://doi.org/10.5281/zenodo.13936406.

16. Ahn N. Rule-based Spanish morphological analyzer built from spell checking lexicon. URL: https://doi.org/10.48550/arXiv.1707.07331.

17. Kessikbayeva G., Cicekli I. Rule based morphological analyzer of Kazakh language. *Proceedings of the 2014 Joint Meeting of SIGMORPHON and SIGFSM*. Baltimore, Maryland, 2014. P. 46–54. URL: https://doi.org/10.3115/v1/W14-2806.

18. Adaptively accelerating Map-Reduce/Spark with GPUs: a case study / K. R. Jayaram et al. *2019 IEEE International Conference on Autonomic Computing (ICAC)*. Umea, Sweden, 2019. P. 105–114. URL: https://doi.org/10.1109/ICAC.2019.00022.

19. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort / N. Satish et al. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. Indianapolis Indiana USA, 2010. P. 351–362. URL: https://doi.org/10.1145/1807167.1807207.

20. Amdahl's law in big data analytics: alive and kicking in TPCx-BB (BigBench) / D. Richins et al. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Vienna, 2018. P. 630–642. URL: https://doi.org/10.1109/HPCA.2018.00060.

21. Karami R., Kao S.-C., Kwon H. Understanding the performance horizon of the latest ML workloads with nonGEMM workloads. URL: https://doi.org/10.48550/arXiv.2404.11788.

22. Borodii I., Osukhivska H. Research on the efficiency of data loading and storage in data lakehouse architectures for the formation of analytical data systems. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2025. No. 4. P. 28–36. URL: https://doi.org/10.32782/it/2025-4-4.
23. Liu H., Huang H. H. SIMD-X: programming and processing of graph algorithms on GPUs. URL: https://doi.org/10.48550/arXiv.1812.04070.
24. Debunking the CUDA myth towards GPU-based AI systems: evaluation of the performance and programmability of Intel's Gaudi NPU for AI model serving / Y. Lee et al. *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. New York, NY, USA, 2025. P. 1760–1776. URL: https://doi.org/10.1145/3695053.3731050.
25. Comparative analysis of large data processing in Apache Spark using Java, Python and Scala / I. Borodii et al. *Proceedings of the 3rd International Workshop on Computer Information Technologies in Industry 4.0 (CITI'2025)*. 2025. URL: https://ceur-ws.org/Vol-4057/paper13.pdf.
26. Fedorovych I., Osukhivska H., Lutsyk N. Performance benchmarking of continuous processing and micro-batch modes in Spark Structured Streaming. *Proceedings of the 4th International Workshop on Information Technologies: Theoretical and Applied Problems (ITTAP'2024)*. 2024. URL: https://ceur-ws.org/Vol-3896/paper5.pdf.
27. Chattaraj R., Chimalakonda S. NLP libraries, energy consumption and runtime: an empirical study. *Proc. ACM Softw. Eng.* 2025. Vol. 2, FSE. P. 2850–2873. URL: https://doi.org/10.1145/3729396.

Історія статті:
Отримано: 16.02.2026 Доопрацьовано: 23.02.2026 Прийнято до друку: 23.03.2026 Опубліковано: 29.03.2026