HetMatch: GPU-Accelerated Heterogeneous String Pattern Matching

Abigale Kim
University of Wisconsin—Madison
Madison, Wisconsin
abigale@cs.wisc.edu

Xiangyao Yu
University of Wisconsin—Madison
Madison, Wisconsin
yxy@cs.wisc.edu

ABSTRACT

String pattern matching is a fundamental operation in data analytics workloads, and appears as a core primitive in data processing libraries and query processing engines. The variable-length nature of strings makes them difficult to process efficiently on GPUs, where peak performance relies on uniform work distribution across threads. In this paper, we present HetMatch, a heterogeneous string pattern-matching technique built directly into NVIDIA's cuDF, that routes strings to different kernels based on their length. HetMatch partitions strings at a 128-byte length threshold, routing short strings to a thread-per-string kernel and long strings to a warp-per-string kernel. Our evaluation across synthetic workloads and 11 real-world datasets shows a geometric mean speedup of 2.5× (up to 13.5×) over cuDF on synthetic workloads, and a geometric mean speedup of 1.3× (up to 2.0×) on real-world datasets.

PVLDB Reference Format:
Abigale Kim and Xiangyao Yu. HetMatch: GPU-Accelerated Heterogeneous String Pattern Matching. VLDB 2026 Workshop: 17th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS26).

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/abigalekim/contains_bench.

1 INTRODUCTION

String data processing in database systems remains a challenging problem. The variable-length nature of strings makes data operators that execute fast on fixed-size data types much less optimal on strings. For example, an equality comparison operator on a numerical datatype is implemented by a single hardware instruction. In contrast, an equality comparison on string data typically requires instructions proportional to the number of characters in the string. As a result, many string operations are computationally intensive. This means that the CPU could be the bottleneck in large-scale text processing. Workloads involving large-scale data processing on strings are also very I/O-intensive, and memory operations are often a bottleneck in database system performance.

The limitations of CPU-based string processing methods motivate a shift toward GPU acceleration. Recently, GPUs have been increasingly adopted for data analytics. Both hardware and software advancements drive this trend. On the hardware side, GPU performance has been increasing more rapidly than that of traditional general-purpose CPUs. The GPU's peak performance and memory bandwidth have increased by over 14× and 4×, respectively, over the past five years alone [24]. On the software side, the hyper-parallelism of GPUs makes them highly applicable to database processing, where processing many rows in parallel is a significant advantage. Overall, both research prototypes [6, 17, 20, 33, 43] and industry systems [5, 8, 14, 26, 28, 37] have demonstrated significant speedups achievable through GPU-based data analytics.

Prior work has shown that GPU acceleration is highly effective for string processing. Several works have demonstrated significant speedups for exact and approximate string pattern matching [4, 15, 16, 18, 21, 31, 35, 44, 45]. These results establish GPUs as a promising platform for string-heavy database workloads. However, pattern-matching performance is suboptimal for variable-length strings, especially when the string length distribution is highly skewed. For example, the parallelization strategy used by NVIDIA's data-frame library cuDF [8] maps each string to either a thread or a warp regardless of its length. This solution can lead to thread divergence and hardware under-utilization, since short strings take less time to process but must wait for the processing of long strings. The current solution in cuDF adaptively determines whether the strings in a column are processed by their thread-per-string or warp-per-string implementation. This approach works when most strings are of similar length, but fails in variable-length cases.

In this paper, we aim to address this issue by introducing a new pattern-matching technique. Specifically, this paper makes the following contributions:

- We design and implement a heterogeneous pattern-matching approach built directly into NVIDIA's cuDF [8]. This approach mitigates thread divergence through a lightweight, length-based index partitioning strategy that routes short strings to the thread-per-string kernel and long strings to the warp-per-string kernel, without introducing excessive overhead.
- We provide a comprehensive evaluation demonstrating that our heterogeneous approach effectively minimizes hardware under-utilization. By benchmarking against existing baseline implementations and cuDF's default execution model, we show maximum speedups of 13.5× on synthetic workloads and 2.0× on real-world datasets.

The rest of the paper is organized as follows: We discuss the GPU architecture and cuDF's pattern-matching algorithm in Section 2. Next, we detail the design and implementation of HetMatch, our heterogeneous string pattern-matching technique, in Section 3.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

Section 4 evaluates the performance of our method against cuDF and other baseline approaches across both synthetic and real-world workloads. Then, we discuss related work in Section 5. Finally, we conclude the paper and outline avenues for future work in Section 6.

2 BACKGROUND

2.1 GPU Architecture

Graphics processing units (GPUs) achieve high throughput through massive parallelism. A GPU consists of many *streaming multiprocessors* (SMs) that execute computation in parallel. Each SM runs one or more *thread blocks* concurrently, where a thread block is a grouping of up to 1024 threads. Within each block, the GPU schedules threads into groups of 32, called *warps*. A warp issues instructions in lockstep, which means all 32 threads execute the same instruction at the same time. This enables the hardware to amortize instruction fetching and scheduling overhead across all threads simultaneously. Peak compute efficiency is achieved when all 32 threads in a warp actively perform useful work throughout execution. When threads within a warp take different control-flow paths due to data-dependent branching, the GPU executes each divergent path sequentially, leaving the remaining threads in the warp idle. This phenomenon, known as *warp divergence*, is a primary source of performance inefficiency for GPU workloads with irregular characteristics.

2.2 cuDF's String Pattern Matching Design

String pattern matching is a core primitive used to implement SQL `LIKE` predicates and determines whether each string in an input column contains a given substring or pattern. cuDF exposes this operation via `contains`, which accepts an input string column and a target pattern. It returns a Boolean array R of the same size as the input column, where $R[i] =$ `true` indicates the pattern is present in string i, and $R[i] =$ `false` indicates it is not.

As shown in Algorithm 1, cuDF calculates the global arithmetic mean length (μ) of all strings in the column. If $\mu \leq 64$ bytes, it dispatches the thread-per-string kernel for the entire string column. Otherwise, it invokes the warp-per-string kernel. These are cuDF's two underlying kernel implementations:

(1) **Thread-Per-String Kernel:** A baseline approach where each CUDA thread sequentially processes a single string. This works efficiently for small, relatively uniform collections of strings, since the execution footprint per warp remains balanced.
(2) **Warp-Per-String Kernel:** An architecture-optimized approach where a warp is assigned to scan a single string concurrently, parsing data one cache line (128 bytes) per iteration to maximize cache efficiency.

While cuDF's design is effective for datasets with a relatively uniform length distribution, it can fail when the string length distribution is highly skewed. When pattern-matching a dataset with high skew, if cuDF chooses the thread-per-string implementation, threads assigned to short strings will sit idle while threads assigned to larger strings are still running. On the other hand, when cuDF dispatches the warp-per-string kernel, warps operating on shorter strings will have some threads sitting idle due to insufficient data.

Algorithm 1 cuDF Contains

Input: String column C, pattern p
Output: Boolean array R
1: $\mu \leftarrow \text{MeanLength}(C)$
2: **if** $\mu \leq 64$ **then**
3: **for all** string $s_i \in C$ **in parallel on GPU do**
4: $R[i] \leftarrow \text{ThreadScan}(s_i, p)$
5: **end for**
6: **else**
7: **for all** string $s_i \in C$ **in parallel on GPU do**
8: $R[i] \leftarrow \text{WarpScan}(s_i, p)$
9: **end for**
10: **end if**
11: **return** R

Algorithm 2 HetMatch Contains

Input: String column C, pattern p
Output: Boolean array R
1: $I \leftarrow \text{NonNullIndices}(C)$
2: $[I_{\text{short}} \mid I_{\text{long}}] \leftarrow \text{Partition}(I, |s_i| \leq 128)$
3: **for all** index $i \in I_{\text{short}}$ **in parallel on GPU do**
4: $R[i] \leftarrow \text{ThreadScan}(C[i], p)$
5: **end for**
6: **for all** index $i \in I_{\text{long}}$ **in parallel on GPU do**
7: $R[i] \leftarrow \text{WarpScan}(C[i], p)$
8: **end for**
9: **return** R

3 HETEROGENEOUS STRING EXECUTION DESIGN

As established in Section 2, cuDF's `contains` implementation selects a single execution strategy, thread-per-string or warp-per-string, for an entire string column based on the average length of the strings in the column. This section presents HetMatch, our heterogeneous string pattern-matching technique, which addresses the hardware under-utilization that arises when a single kernel strategy is applied to a column with high length skew.

3.1 Motivation

The core inefficiency of single-strategy dispatch is warp divergence caused by string length skew. On skewed workloads, cuDF averages only 16.8 active threads per instruction, approximately 50% of the theoretical maximum of 32. We quantify this further in Section 4.3.3.

Our approach addresses this by partitioning string indices by length before dispatch, routing short strings to the thread-per-string kernel and long strings to the warp-per-string kernel. By routing only short strings to the thread-per-string kernel, warp divergence is bounded by the variance among short strings, which is small by construction. Long strings, meanwhile, are handled by the warp-per-string kernel, which amortizes memory access cost across 32 threads by reading string data one cacheline at a time. Sections 3.2–3.4 describe each phase in detail.

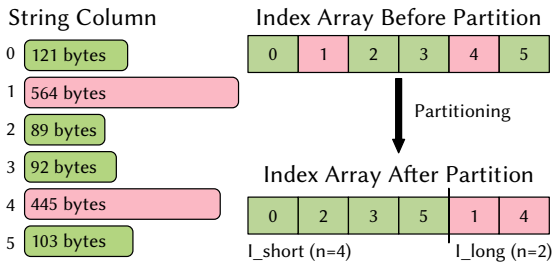

Figure 1: Overview of Index Partitioning in HetMatch

3.2 Index Partitioning

In the index partitioning phase, we split the string indices into two contiguous groups based on a 128-byte length threshold. cuDF applies a similar length-based heuristic with a threshold of 64 bytes [38], where prior benchmarking showed that the thread-per-string kernel is more performant below that threshold. We raise the threshold to 128 bytes to align with the warp-per-string kernel's memory access granularity. Since the warp-per-string kernel reads string data in 128-byte chunks, strings shorter than this threshold incur wasted work within each load. Routing these strings to the thread-per-string path avoids this overhead entirely.

Figure 1 illustrates the partitioning process on a small example column. As shown, the string column contains strings of mixed lengths and is never modified; only the index array is reordered. Before partitioning, the index array stores indices in their original order, interleaving short and long string indices. The partitioning algorithm reorders the index array in-place such that all indices corresponding to short strings (≤128 bytes) occupy the left region I_{short} and all indices corresponding to long strings (>128 bytes) occupy the right region I_{long}. We then pass a pointer to the start of the array into the thread-per-string kernel and a pointer to the first long string index into the warp-per-string kernel, each paired with its corresponding element count. We considered accessing the offset array directly to determine string lengths, but found the extra indirection introduced measurable overhead compared to operating on the index array with an explicit predicate.

Partitioning is implemented using Thrust's `partition` [27], which operates in-place and is unstable by default. We evaluated two designs: an unstable partition (Thrust) and a stable partition (CUB [23]), which preserves relative element order and improves spatial locality when accessing string data during kernel execution. In practice, the locality benefit of stable partitioning was insufficient to offset its greater algorithmic overhead, and the unstable variant delivered consistently better end-to-end performance across all benchmarks.

3.3 Thread-Per-String Execution

After partitioning, the thread-per-string kernel processes the short strings by iterating over I_{short}. Each CUDA thread with global thread ID t is assigned one string, reads its assigned string index from $I_{\text{short}}[t]$, extracts the corresponding string from the column, and performs a sequential scan for the target pattern. This naïve design performs well for short strings because the per-thread work is small and relatively uniform, leading to low divergence within a warp.

3.4 Warp-Per-String Execution

The warp-per-string kernel processes strings longer than 128 bytes. Similarly to the thread-per-string kernel, in the warp-per-string kernel, each CUDA warp with global warp id w is assigned the string with index $I_{\text{long}}[w]$. Then, the warp's 32 threads cooperate to scan each string 4 bytes at a time, processing one full cache line of string data per iteration step. This cooperative access pattern improves both memory throughput and cache hit rates for long strings.

4 EVALUATION

In this section, we evaluate HetMatch across synthetic and real-world workloads to assess end-to-end performance, partitioning overhead, and the relationship between string length distribution and performance gains. The rest of this section is organized as follows: we discuss the experimental setup in Section 4.1, explain the synthetic and real world benchmarks in Section 4.2, evaluate performance on synthetic datasets in Section 4.3, and evaluate performance on real-world datasets in Section 4.4.

4.1 Experimental Setup

All experiments were conducted on a machine equipped with an NVIDIA H100 PCIe GPU with 80 GB of device memory and dual Intel Xeon Gold 6530 processors (32 cores, 64 threads each) with 240 GB of CPU RAM. The system runs Ubuntu 22.04 and CUDA 13.0. Each experiment reports the median execution time over five trials, excluding the first cold run to account for memory overhead.

We compare against the following baseline implementations:

- **Thread-per-String**: A GPU kernel that always assigns a single thread per string for pattern matching.
- **Warp-per-String**: A GPU kernel that always assigns a warp (32 threads) per string for pattern matching.
- **cuDF 25.10** [8]: NVIDIA's production GPU dataframe library, which dispatches a thread-per-string kernel when the mean string length of the input is less than or equal to 64 bytes, and a warp-per-string kernel otherwise [38]. This serves as our primary GPU baseline.
- **`DataFrame`** [12]: A multi-threaded C++ dataframe library that parallelizes string operations across CPU cores.
- **Pandas** [30]: A single-threaded Python dataframe library.

4.2 Workloads

We evaluate HetMatch on both synthetic and real-world workloads, chosen to cover a range of string length distributions that stress the partitioning decision. All workloads use the `contains` substring matching predicate described in Section 2.2. Synthetic workloads allow us to precisely control skew, while real-world datasets validate that the performance benefits extend to practical settings where skew arises naturally.

4.2.1 Synthetic Workloads. We use two synthetic workload types, both designed to reflect heterogeneous length profiles that stress

Dataset	No. Strings (millions)	≤128 bytes	>128 bytes	Mean (bytes)	Median (bytes)	95th Pct. (bytes)	99th Pct. (bytes)
Facebook Comments [13]	63.6	59.3%	40.7%	169	98	490	1218
Facebook Posts [13]	92.8	63.0%	37.0%	116	104	256	369
Reddit Comments [1]	63.3	64.5%	35.5%	174	85	610	1257
Twitter Posts [32]	144.9	88.9%	11.1%	74	69	136	141
Amazon Reviews [2]	71.2	63.0%	37.0%	151	89	484	982
Yelp Reviews [39]	18.9	7.7%	92.3%	566	405	1559	2633
GitHub Commit Messages [10]	25.0	30.9%	69.1%	430	280	1300	2563
Common URLs [7]	140.0	92.1%	7.9%	77	65	147	256
Shopify Product Names [34]	179.9	91.0%	9.0%	60	47	168	198
NUS Text Messages [22]	133.5	76.8%	23.2%	81	62	161	276
LLM Chat Logs [19]	15.1	38.9%	61.1%	711	263	3177	5132

Table 1: Statistics on Real World Datasets

the partitioning decision. The first is a *skewed* workload: distribution (a, b) consists of 99.5% of strings with length a bytes, with the remaining strings having length b. The second is a *bimodal* workload: distribution $(\mu_1, \sigma_1, \mu_2, \sigma_2)$ consists of 99.5% of strings drawn from a normal distribution (μ_1, σ_1) and the rest from (μ_2, σ_2), where $\mu_2 > \mu_1$. Both workload types are asymmetric by design. Since the cost mismatch between thread-per-string and warp-per-string processing grows with heterogeneity, the benefit of routing strings to the appropriate kernel becomes more pronounced.

4.2.2 Real-World Workloads.

Table 1 summarizes the statistics of our eleven real-world datasets. We selected datasets that exhibit natural length skew, a common feature of real-world textual data. Many of our datasets are right-skewed, with most strings falling below 128 bytes. For example, the Twitter Posts (88.9%), Common Crawl URLs (92.1%), and Shopify Product Names (91.0%) datasets are dominated by short strings. Others have fewer very short strings but exhibit longer tails. For instance, the LLM Chat Logs dataset has only 38.9% of strings under 128 bytes, but a 95th percentile length of 3,177 bytes, and the GitHub Commit Messages dataset exhibits a similar distribution (30.9% of strings under 128 bytes, 95th percentile length of 1,300 bytes). Across all datasets, the median is consistently below the mean, confirming the presence of skewed length distributions well-suited to stress the partitioning decision.

4.3 Evaluation on Synthetic Datasets

We evaluate our heterogeneous implementation on synthetic skewed and bimodal workloads designed to stress the partitioning decision.

4.3.1 End-to-End Performance. Figure 2 presents the end-to-end execution time of all implementations normalized to cuDF across the synthetic skewed and bimodal workloads. HetMatch achieves a geometric mean speedup of 2.5× over cuDF, with a peak speedup of 13.5×. These results show that HetMatch is consistently faster on workloads with highly skewed length distributions. Our performance advantage over cuDF is most pronounced on bimodal workloads. In bimodal workloads where the mean string length is less than 64 bytes, cuDF dispatches the thread-per-string kernel, and the presence of long strings incurs significant overhead under its single-kernel strategy. Additionally, all GPU implementations are significantly faster than both CPU baselines. Since Pandas is single-threaded and the C++ dataframe library is bound by the number of available CPU cores, neither can exploit the degree of parallelism available on the GPU for large-scale string processing.

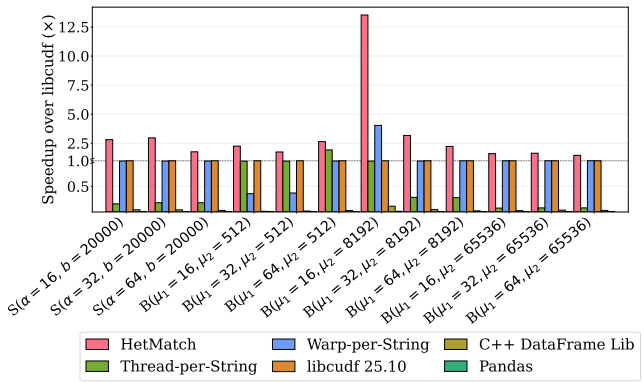

Figure 2: Speedup Against CPU Implementations and cuDF on Synthetic Workloads

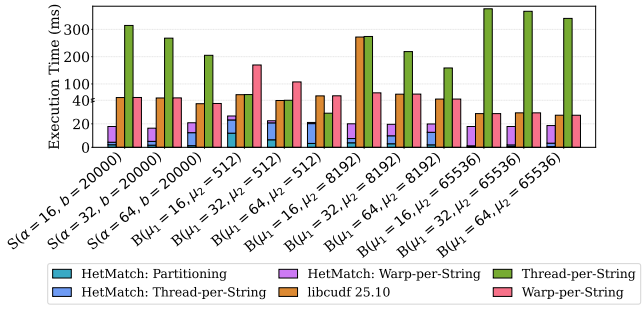

Figure 3: Breakdown of Execution Time on Synthetic Workloads

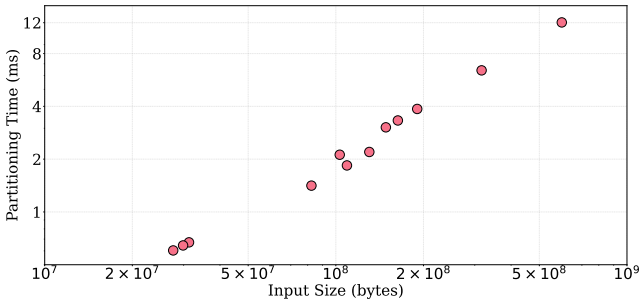

Figure 4: Input Size vs. Partitioning Time on Synthetic Workloads

Figure 3 breaks down the execution time of HetMatch into its three constituent phases: partitioning, thread-per-string execution, and warp-per-string execution. Across nearly all workloads, the partitioning cost is negligible relative to the kernel execution phases. Additionally, the combined kernel execution time is consistently lower, confirming that the performance gains of our approach stem from more efficient kernel dispatch.

4.3.2 Cost of Partitioning. In this section, we evaluate the computational overhead of the partitioning phase relative to total query

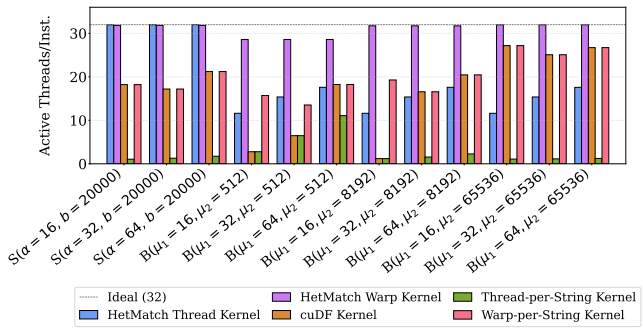

Figure 5: Average Number of Threads Executing per Instruction on Synthetic Workloads

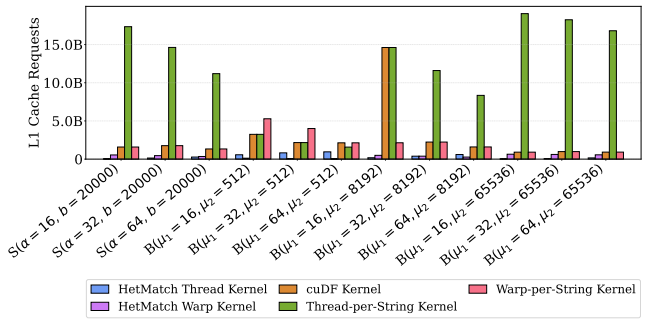

Figure 6: Total L1 Requests on Synthetic Workloads

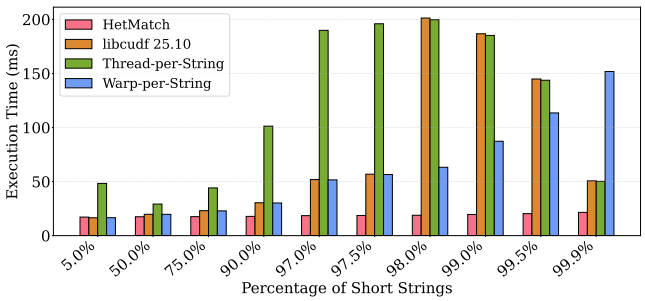

Figure 7: Relationship between Percentage of Short Strings and Execution Time

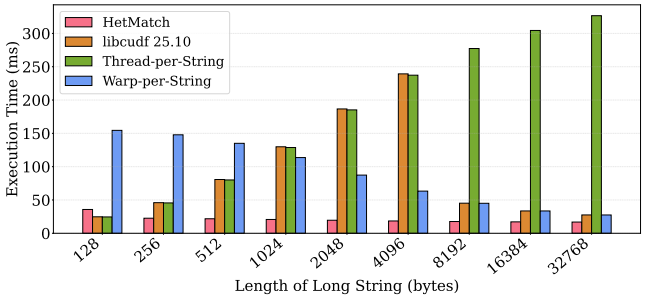

Figure 8: Relationship between Length of Long Strings and Execution Time

execution time. Figure 3 shows that across most datasets, the partitioning phase accounts for a small fraction of the total execution time. The exception is the bimodal (μ_1=16, σ_1=5, μ_2=512, σ_2=100) dataset, where the partitioning cost is proportionally higher because the distribution is dominated by very short strings. This means that there are many more strings to partition relative to the total bytes that need to be scanned during matching. To understand why partitioning remains cheap in general, Figure 4 plots the absolute partitioning time as a function of total input size in bytes, showing that partitioning time scales linearly with input size. This aligns with the algorithmic complexity of the partitioning step, which reads only the string offset array to evaluate string lengths and therefore runs in $O(n)$ time where n is the number of strings. String matching must process the character data itself, giving a lower-bound complexity of $O(C)$ where C is the total number of characters. Since $n \leq C$ in all cases, and usually $n \ll C$, the partitioning overhead remains strictly bounded below the minimum total execution cost of the matching phase across every workload we tested.

4.3.3 Performance Breakdown against cuDF. We run a performance analysis on both HetMatch and cuDF to show why our implementation outperforms cuDF, using NSight Compute [25] to capture kernel metrics for the pattern-matching kernels across our synthetic workloads. We find that our implementation has significantly less warp divergence than cuDF, which occurs when threads in a single warp take different execution paths. Figure 5 shows the average number of active threads executing per instruction for both our kernels and cuDF's kernel. Our thread-per-string and warp-per-string kernels average 20.2 and 29.6 active threads per instruction, respectively, compared to 16.8 for cuDF. The lower thread utilization in cuDF arises from its single-kernel dispatch strategy. When cuDF dispatches its thread-per-string kernel, threads assigned to shorter strings will complete early and sit idle, while threads assigned to longer strings will continue executing for a while. On the other hand, when cuDF dispatches its warp-per-string kernel, warps operating on strings shorter than 128 bytes will have threads doing no useful work due to insufficient data. Figure 6 shows the L1 cache requests made during each workload. HetMatch reduces L1 cache requests by up to 21.6×, with an average reduction of 3.8× over cuDF. Since every warp launch incurs memory requests to retrieve string data, and these requests are not proportional to the number of strings processed, the more targeted kernel dispatch in our implementation results in substantially fewer memory requests.

4.3.4 Relationship Between Data Skew to Performance. Our next set of experiments examines how dataset skew affects HetMatch, and identifies the distributions that benefit most from our technique. In the first experiment, we measure the average execution time of HetMatch and cuDF on skewed datasets with short strings of 16 bytes and long strings of 2048 bytes, varying the fraction of short strings from 5% to 99.9%. Figure 7 plots the average execution time as this fraction varies. The best speedup over cuDF occurs when the mean string length is less than 64 bytes, causing cuDF to dispatch the thread-per-string kernel and incur overhead when processing

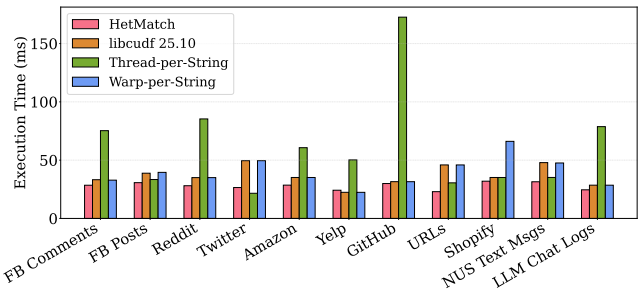

Figure 9: End-to-End Performance on Real World Workloads

the longer strings. As the fraction of long strings increases, cuDF switches to the warp-per-string kernel. At this point, performance converges because the work on the 16-byte strings becomes constant across executions, and the only remaining variation is in the processing of the longer strings. Nevertheless, HetMatch outperforms cuDF for all distributions with 50% or more long strings, demonstrating that our approach remains beneficial even under moderate skew. In the second experiment, we fix the dataset at 99% short (16-byte) strings and 1% long strings, and vary the length of the long strings from 128 to 32,768 bytes. Figure 8 shows the execution times of HetMatch and cuDF on these datasets. When the long strings are short enough that the mean falls below cuDF's 64-byte dispatch threshold, HetMatch achieves speedup by routing the long strings to the warp-per-string kernel while cuDF applies thread-per-string uniformly. Once the mean exceeds the threshold and cuDF switches to warp-per-string, HetMatch still achieves approximately 2× speedup by avoiding wasted work on the short strings. Together, these results demonstrate that our technique delivers consistent benefit across a wide range of skew profiles.

4.4 Evaluation on Real World Datasets

We evaluate HetMatch on eleven real-world datasets (Table 1) to demonstrate that the performance benefits extend beyond synthetic workloads. While real-world datasets do not exhibit the extreme skew of our synthetic stress tests, they naturally contain a mix of short and long strings that reflects the heterogeneity encountered in practice. We describe these datasets in more detail in Section 4.2.2. Figure 9 shows the speedup of our heterogeneous implementation over cuDF across all real-world datasets. We achieve a geometric mean speedup of 1.3× and a maximum speedup of 2.0×. Although skew in real-world datasets is less pronounced than in our synthetic workloads, it is sufficient for our approach to deliver meaningful benefits. Two distinct sources of improvement contribute to these gains. First, in some datasets, the mean string length is skewed upward by a small number of long strings, causing cuDF to dispatch the warp-per-string kernel and incur wasted work on the short strings that dominate the distribution. For example, we achieve a speedup of 1.9× on the Twitter Posts dataset and 2.0× on the Common URL dataset due to this effect. Second, in datasets where neither the thread-per-string nor warp-per-string kernel is uniformly well-suited to the length distribution, HetMatch outperforms both single-kernel baselines by routing each string to the appropriate kernel. For example, on the Amazon Reviews dataset, we achieve a speedup of 1.2× over the pure warp-per-string approach and 2.1× over the pure thread-per-string approach. Even if cuDF chose the more optimal approach, HetMatch outperforms cuDF because neither single kernel approach is well-suited to the full range of string lengths present in the dataset.

5 RELATED WORK

Accelerating analytical query processing on GPUs has been extensively studied, spanning both research prototypes [6, 17, 33, 40–43] and production systems [5, 14, 26, 28, 37]. These efforts have demonstrated significant speedups over CPU-based systems by exploiting the GPU's high memory bandwidth and massive parallelism. String pattern matching is a first-class primitives in GPU dataframe libraries like cuDF [8], which serve as the backend for systems including Polars [11] and SiriusDB [41].

Prior work has implemented a range of string matching algorithms on GPUs, including multi-threaded brute-force, Quick-Search [36], Rabin-Karp, Aho-Corasick, KMP, Boyer-Moore, and bit-parallel scan methods [3, 4, 16, 18, 21, 35, 44, 45], demonstrating significant speedups over CPU implementations. Across the previous literature and the many different algorithms implemented, we find that memory access efficiency and high thread utilization are significant contributors to performance. Additionally, we find that these works use optimizations such as shared memory, coalesced memory access patterns, and alternative string layouts to improve memory throughput. However, all of these works adopt a fixed parallelism granularity in their implementations and do not address workloads with high skew length distributions.

The limitation of fixed parallelism granularity is an instance of a broader challenge in GPU computing. Since irregular workloads introduce warp divergence when per-thread work varies with input data, adaptive execution strategies can yield significant performance gains. Osama et al. [29] propose a general load-balancing abstraction that decouples scheduling from work processing for irregular GPU algorithms, and Seer [9] uses a learned decision tree to select among kernels at runtime for irregular problems such as sparse matrix vector multiplication (SpMV). These works show that GPU execution on irregular workloads benefits from dynamic kernel selection strategies. We draw on this insight to motivate our heterogeneous dispatch approach for processing string data with high length skew.

6 CONCLUSION AND FUTURE WORK

In this paper, we present HetMatch, a heterogeneous string pattern-matching approach built into NVIDIA's cuDF. Our approach reduces the warp divergence caused by cuDF's single-kernel dispatch strategy, achieving a geometric mean speedup of 2.5× on synthetic workloads, peaking at 13.5×, and a geometric mean speedup of 1.3× on real-world datasets, peaking at 2.0×. Future work includes extending the heterogeneous dispatch model to other string primitives, such as regex matching and string updating.

ACKNOWLEDGMENTS

We thank Lambda Labs and eFabric for providing the cloud computing resources that supported this research. We also thank the reviewers for their insightful feedback.

REFERENCES

[1] 1 million Reddit comments from 40 subreddits 2026. https://www.kaggle.com/datasets/smagnan/1-million-reddit-comments-from-40-subreddits.
[2] Amazon Reviews 2023 2023. https://amazon-reviews-2023.github.io/.
[3] Saman Ashkiani, Nina Amenta, and John D. Owens. 2016. Parallel Approaches to the String Matching Problem on the GPU. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (Pacific Grove, California, USA) *(SPAA '16)*. Association for Computing Machinery, New York, NY, USA, 275–285. https://doi.org/10.1145/2935764.2935800
[4] M. Musthafa Baig, S. Sivakumar, and Soumya Ranjan Nayak. 2020. Optimizing Performance of Text Searching Using CPU and GPUs. In *Progress in Computing, Analytics and Networking*, Himansu Das, Prasant Kumar Pattnaik, Siddharth Swarup Rautaray, and Kuan-Ching Li (Eds.). Springer Singapore, Singapore, 141–150.
[5] BlazingSQL 2025. https://github.com/BlazingDB/blazingsql
[6] Sebastian Breß. 2014. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14, 3 (2014), 199–209. https://doi.org/10.1007/s13222-014-0164-z
[7] Common Crawl 2025. https://data.commoncrawl.org/.
[8] cuDF—GPU DataFrames 2025. github.com/rapidsai/cudf
[9] Leon Frenot and Fernando Magno Quintão Pereira. 2024. Reducing the Overhead of Exact Profiling by Reusing Affine Variables. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24)*. ACM, 150–161. https://doi.org/10.1145/3640537.3641569
[10] GitHub Commit Messages Dataset 2026. https://www.kaggle.com/datasets/dhruvildave/github-commit-messages-dataset.
[11] GPU Support - Polars use guide 2026. https://docs.pola.rs/user-guide/lazy/gpu/
[12] hosseinmoein/DataFrame C++ Library 2026. https://github.com/hosseinmoein/DataFrame.
[13] jbencina/facebook-news 2026. https://github.com/jbencina/facebook-news.
[14] Kinetica 2025. https://www.kinetica.com/
[15] Charalampos Kouzinopoulos, Panagiotis Michailidis, and Konstantinos G. Margaritis. 2015. Multiple string matching on a GPU using CUDA. *Scalable Computing: Practice and Experience* 16 (06 2015). https://doi.org/10.12694/scpe.v16i2.1085
[16] Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis. 2009. String Matching on a Multicore GPU Using CUDA. In *2009 13th Panhellenic Conference on Informatics*. 14–18. https://doi.org/10.1109/PCI.2009.47
[17] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331
[18] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu. 2010. Accelerating String Matching Using Multi-Threaded Algorithm on GPU. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. 1–5. https://doi.org/10.1109/GLOCOM.2010.5683320
[19] LMSYS-Chat-1M 2023. https://huggingface.co/datasets/lmsys/lmsys-chat-1m.
[20] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705
[21] Yasuaki Mitani, Fumihiko Ino, and Kenichi Hagihara. 2017. Parallelizing Exact and Approximate String Matching via Inclusive Scan on a GPU. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 1989–2002. https://doi.org/10.1109/TPDS.2016.2645222
[22] NUS SMS Corpus 2026. https://www.kaggle.com/code/dorothygao1023/nus-sms-corpus/.
[23] NVIDIA CUB library devicePartition implementation 2026. https://github.com/NVIDIA/cccl/blob/875a7ce6e8c16d9de67b794922152fd54cd75048/cub/cub/device/device_partition.cuh.
[24] NVIDIA DGX B300 GPU 2026. https://www.nvidia.com/en-us/data-center/dgx-b300/.
[25] NVIDIA Nsight Compute 2026. https://developer.nvidia.com/nsight-compute.
[26] NVIDIA RAPIDS 2025. https://rapids.ai
[27] NVIDIA Thrust library partition implementation 2026. https://github.com/NVIDIA/cccl/blob/875a7ce6e8c16d9de67b794922152fd54cd75048/thrust/thrust/system/cuda/detail/partition.h.
[28] OmniSci 2025. https://omnisci.com
[29] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. 2023. A Programming Model for GPU Load Balancing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23)*. ACM, 79–91. https://doi.org/10.1145/3572848.3577434
[30] pandas 2026. https://pandas.pydata.org/.
[31] Michael C. Schatz and Cole Trapnell. 2011. Fast Exact String Matching on the GPU. https://api.semanticscholar.org/CorpusID:14444749
[32] Sentiment140 dataset with 1.6 million tweets 2026. https://www.kaggle.com/datasets/kazanova/sentiment140.
[33] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595
[34] Shopify Product Catalogue 2026. https://huggingface.co/datasets/Shopify/product-catalogue.
[35] Evangelia A. Sitaridi and Kenneth A. Ross. 2016. GPU-accelerated string matching for database applications. *The VLDB Journal* 25, 5 (2016), 719–740. https://doi.org/10.1007/s00778-015-0409-y
[36] Daniel M. Sunday. 1990. A very fast substring search algorithm. *Commun. ACM* 33, 8 (Aug. 1990), 132–142. https://doi.org/10.1145/79173.79184
[37] The RAPIDS Accelerator for Apache Spark 2025. https://nvidia.github.io/spark-rapids
[38] Use warp per string for long strings in cudf::strings::contains() 2026. https://github.com/rapidsai/cudf/pull/10739.
[39] Yelp Dataset 2026. https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset.
[40] Bobbi Yogatama, Weiwei Gong, and Xiangyao Yu. 2024. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endow.* 17, 13 (Sept. 2024), 4709–4722. https://doi.org/10.14778/3704965.3704977
[41] Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei Gong, and Xiangyao Yu. 2026. Rethinking Analytical Processing in the GPU Era. In *CIDR 2026, Conference on Innovative Data Systems Research*. https://arxiv.org/abs/2508.04701
[42] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (July 2022), 2491–2503. https://doi.org/10.14778/3551793.3551809
[43] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. https://doi.org/10.14778/2536206.2536210
[44] Xinyan Zha and Sartaj Sahni. 2011. Multipattern string matching on a GPU. In *2011 IEEE Symposium on Computers and Communications (ISCC)*. 277–282. https://doi.org/10.1109/ISCC.2011.5983790
[45] Xinyan Zha and Sartaj Sahni. 2013. GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU. *IEEE Trans. Comput.* 62, 6 (2013), 1156–1169. https://doi.org/10.1109/TC.2012.61