HYDRA: Modernizing Single-GPU MapReduce Execution

Huyao Yang Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
yanghuyao@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—MapReduce is a widely used programming model for data processing, and its single-node execution efficiency remains important as data volumes and workload intensity continue to grow. Although prior work has leveraged GPUs to accelerate MapReduce workloads, legacy implementations do not fully exploit contemporary GPU software and memory mechanisms.**

In this paper, we present HYDRA, a compatibility-preserving modernization of a single-GPU, in-memory MapReduce execution engine. HYDRA jointly redesigns the Shuffle and data-movement paths while preserving the original programming interface and semantics. Specifically, HYDRA 1) replaces the legacy bitonic sort implementation with a maintained, Thrust-backed parallel merge-sort path; 2) encodes record boundaries using compact end offsets, reducing per-record offset metadata from 16 to 8 bytes; and 3) introduces an alignment-aware vectorized copy path with a scalar fallback for short or irregular records.

We implement HYDRA in CUDA on top of Mars and evaluate it using eight representative MapReduce workloads. Compared with Mars under the same experimental conditions, HYDRA achieves a geometric mean speedup of 2.51× and a maximum speedup of 9.40×. Ablation results identify the Thrust-backed parallel merge-sort path as the primary source of speedup for shuffle-intensive workloads, while metadata compression and vectorized copying provide workload-dependent benefits. These results demonstrate that substantial performance gains can be achieved in a legacy GPU MapReduce engine through a small set of semantics-preserving execution-path changes.

***Index Terms*—GPU computing, MapReduce, Heterogeneous programming**

I. INTRODUCTION

MapReduce is a widely used programming model for processing data across diverse domains [1] [2] [3] [4] [5]. While MapReduce can coordinate work across multiple nodes, the efficiency of each node remains important to end-to-end performance. This paper focuses specifically on modernizing the execution path of a single-node, single-GPU MapReduce engine.

Recognizing this challenge, researchers have explored GPU acceleration to improve the performance of single-node computation. In particular, prior work [6] offloads the Map and Reduce phases to GPUs by implementing them as GPU kernels using low-level programming interfaces (*e.g.*, CUDA, HIP, or OpenCL). To mitigate performance bottlenecks in MapReduce workloads, such as memory-bound shuffling, prior studies [6] [7] adopt a range of techniques, including atomic-free output aggregation, optimized data layouts, and customized execution pipelines that exploit the GPU memory hierarchy and massive thread parallelism. Furthermore, to enhance the flexibility, scalability, and performance of GPU-based distributed computing infrastructures, prior work [8] proposes configurable execution models that replace the rigid map-shuffle-reduce paradigm with user-defined function call graphs, thereby enabling more efficient data staging and reducing inter-kernel communication overhead. Other studies [9] [10] [11] [12] further extend single-GPU acceleration to cluster environments through MPI or integration with big data processing infrastructures such as Spark [13] [14] and Hadoop [5] [15], enabling acceleration that is compatible with existing data-processing ecosystems.

These advances, while valuable, do not fully address the efficient execution of legacy single-GPU MapReduce engines. Some earlier implementations do not account for characteristics of contemporary GPU software and memory systems, leading to suboptimal data structures and memory access patterns that degrade performance on contemporary hardware. In particular, data movement across the CPU-GPU boundary can add communication overhead. For example, Mars [6] transfers intermediate data between the Map and Reduce phases using verbose metadata formats, which inflate PCIe traffic without providing commensurate computational benefits. More critically, existing approaches often neglect the importance of memory access patterns in MapReduce workloads, resulting in uncoalesced memory accesses and underutilization of GPU memory bandwidth. For example, StreamMR [7] relies on coordination mechanisms based on atomic operations that induce inefficient memory access patterns on AMD GPUs.

In this paper, we present HYDRA, a GPU-accelerated MapReduce execution engine that modernizes Mars' single-GPU execution path. We examine the combination of algorithmic choices, data representations, and memory access strategies for MapReduce workloads on a contemporary CUDA software stack. This modernization addresses three implementation challenges.

The first challenge (**C1**) lies in overcoming the inefficiency of hand-tuned sorting kernels. These ad hoc implementations are error-prone, difficult to maintain, and difficult to adapt to evolving software stacks and diverse data distributions.

* The corresponding author.

To address this challenge, we invoke `thrust::sort` with the existing user-defined comparator; in our CUDA/Thrust configuration, this selects Thrust's parallel merge-sort path [16]. This approach eliminates error-prone low-level implementations, supports arbitrary user-defined comparators, and preserves the interface required by the Shuffle stage.

The second challenge (**C2**) concerns minimizing CPU-GPU communication overhead, which arises from frequent data transfers, synchronization costs, and limited PCIe bandwidth in heterogeneous computing environments. The impact of this overhead on end-to-end performance depends on the metadata ratio and execution profile of each workload.

To address this challenge, we redesign the end-to-end MapReduce execution pipeline to reduce redundant data movement. In particular, HYDRA employs a compact representation of communication metadata based on end offsets. Rather than transferring both a start offset and a size for each reducer segment, HYDRA communicates only a single end offset, thereby halving the volume of offset metadata transferred over PCIe. This representation requires only lightweight boundary reconstruction: each GPU thread reconstructs segment boundaries by reading two adjacent offsets, interpreted as the previous and current end offsets, thus preserving the original MapReduce semantics.

The third challenge (**C3**) arises from inefficient intra-GPU data movement during MapReduce execution. In particular, fine-grained key–value copying and intermediate buffer reorganization rely heavily on byte-wise scalar memory operations, which underutilize GPU memory bandwidth and introduce unnecessary instruction and control overhead. This inefficiency becomes especially pronounced in shuffle-intensive workloads where data movement, rather than computation, dominates execution time.

To address this challenge, HYDRA introduces *Copy Vectorization*, an intra-GPU data-movement optimization that replaces byte-wise scalar copies with word-aligned, 4-byte-wide memory operations whenever possible. When copy sizes and alignment permit, the vectorized path uses fewer, wider memory operations than byte-wise copying. The performance benefit depends on the workload. The optimization preserves the MapReduce programming interface and semantics.

We implemented a software prototype of HYDRA with CUDA and evaluated it on eight representative MapReduce workloads, with input sizes ranging from 22 MB to 1.75 GB. Compared with Mars under the same experimental setup, HYDRA achieves a geometric mean speedup of 2.51× and a maximum speedup of 9.40×. In summary, this paper makes the following contributions:

- We revisit single-GPU MapReduce execution on a contemporary CUDA software stack and examine the mismatch between a legacy execution pipeline and modern GPU primitives and memory-access mechanisms.
- We design and implement HYDRA, a compatibility-preserving modernization of Mars that integrates a maintained GPU sorting primitive, a compact end-offset metadata representation, and alignment-aware vectorized copying without changing the MapReduce programming interface or semantics.
- We evaluate HYDRA on eight representative workloads using end-to-end measurements and ablation studies. HYDRA achieves a geometric mean speedup of 2.51× and a maximum speedup of 9.40× over Mars, while the results show that the effects of metadata compression and copy vectorization are workload dependent.

We publicly release the HYDRA source code, benchmark suite, and evaluation artifacts to support open science and reproducibility within the research community: https://doi.org/10.5281/zenodo.18438595.

II. BACKGROUND

A. MapReduce

MapReduce [17] is a widely adopted programming model for large-scale data processing that abstracts parallel computation into two user-defined functions:

$$Map : (k_1, v_1) \rightarrow list(k_2, v_2)$$
$$Reduce : (k_2, list(v_2)) \rightarrow list(k_2, v_3)$$

The Map function processes input key-value pairs and emits intermediate key-value pairs, which are subsequently shuffled by key and passed to the $Reduce$ function to generate final aggregated results. In practice, a MapReduce job partitions input data into independent chunks that can be processed concurrently by mappers. The mapper outputs then undergo a shuffle phase, during which data are sorted and redistributed by key, before being consumed by reducers.

This programming model enables automatic parallelization, fault tolerance, and distribution across clusters of commodity machines, and has therefore been widely applied in domains such as data mining [18], machine learning [4], and bioinformatics [5].

While early implementations (*e.g.*, Hadoop [19]) prioritized distributed reliability over single-node efficiency, subsequent work has focused on accelerating MapReduce on shared-memory systems [20] [21] [22], multi-core CPUs, and more recently, GPUs [13] [23] [10] [18] [14] [5].

B. GPU Architectures

Graphics processing units (GPUs) are specialized hardware accelerators designed for massively parallel computation. Unlike CPUs, which prioritize low-latency sequential execution, GPUs can execute thousands of threads concurrently by integrating a large number of arithmetic logic units organized into streaming multiprocessors (or compute units in AMD's terminology) [24]. This architectural design makes GPUs particularly well suited for data-parallel workloads commonly found in scientific computing, machine learning, and large-scale data processing.

To harness the computational power of GPUs, general-purpose parallel programming models and languages (*e.g.*, CUDA or HIP [24]) have been introduced. These programming models are heterogeneous, comprising host components that

run on CPUs and *kernels* that execute in parallel on GPUs. Kernels follow a single-program-multiple-data (SPMD) style, in which computation is organized as grids of thread blocks, with each block containing multiple cooperating threads that execute the same code on different data elements. Threads within a block execute in a single-instruction-multiple-thread (SIMT) manner: they are scheduled onto streaming multiprocessors at warp granularity and execute in lockstep, thereby enabling high computational throughput.

III. Design

In this section, we present the design of Hydra. We begin with an overview of the workflow (§ III-A), then detail our approach to metadata compression (§ III-B), the sorting algorithm (§ III-C), and copy vectorization (§ III-D). Finally, we introduce the implementation of Hydra (§ III-E).

A. Overview

Fig. 1 presents an overview of Hydra's workflow, which extends the standard MapReduce pipeline with GPU-specific optimizations at every stage. First, in the PCIe Transfer stage, we reduce CPU-GPU communication costs by compressing MapReduce metadata. Specifically, we encode reducer boundaries using compact end offsets, halving the volume of offset metadata transferred over PCIe while enabling lightweight boundary reconstruction on the GPU. Second, in the Map stage, we execute user-defined map functions entirely on the GPU and apply copy vectorization to accelerate the writing of intermediate key-value pairs via word-aligned, coalesced memory accesses. Next, in the Shuffle stage, we address sorting inefficiency by replacing hand-tuned bitonic sort kernels [25] with Thrust's maintained parallel merge-sort path. This change modernizes the Shuffle implementation while preserving its comparison interface. Finally, in the Reduce stage, we aggregate values in parallel and use copy vectorization during input loading and result emission to minimize intra-GPU data movement and maximize memory bandwidth utilization.

B. Metadata Compression

In this section, we describe how we reduce CPU-GPU communication to accelerate GPU-based MapReduce execution.

We observe that GPU execution typically follows a host-driven model, in which the device serves as an accelerator and data must move across the PCIe bus. We further note that GPU-accelerated MapReduce frameworks follow the same pattern: we must transfer input data from the host to the GPU before executing Map, Shuffle, and Reduce operations on the device. For large-scale workloads, we find that these host-device transfers can dominate end-to-end performance, motivating us to reduce PCIe traffic through metadata compression.

To minimize this overhead, we compress data before transfer and use GPU parallelism to decompress it on the device. Specifically, MapReduce records are of the form {key, value, offset}, where offset is a quadruple {keyStart, keySize, valueStart, valueSize}. We replace this full offset representation with a compact prefix-sum pair {keyEnd, valueEnd}, halving offset metadata storage requirements and reducing PCIe traffic. On the GPU, we reconstruct the original quadruple using a lightweight parallel decompression process, as shown in Algorithm 1. We reduce the volume of data transferred over PCIe without significantly increasing overhead in other stages.

Algorithm 1: Metadata Compression Pipeline

Input: Offset array $\mathcal{O} = \{(k_s^i, k_l^i, v_s^i, v_l^i)\}_{i=0}^{N-1}$
Output: Compressed array $\mathcal{C}$ or reconstructed offsets

Function `HostCompress(`$\mathcal{O}$`)`**:**
  for $i \leftarrow 0$ **to** $N-1$ **do**
    $k_e^i \leftarrow k_s^i + k_l^i$;
    $v_e^i \leftarrow v_s^i + v_l^i$;
  return $\mathcal{C} = \{(k_e^i, v_e^i)\}_{i=0}^{N-1}$;

Function `DeviceDecompress(`$\mathcal{C}$`)`**:**
  $k_e^{-1} \leftarrow 0$, $v_e^{-1} \leftarrow 0$; `// Boundary condition`
  for $i \leftarrow 0$ **to** $N-1$ **do**
    $k_s^i \leftarrow k_e^{i-1}$;
    $k_l^i \leftarrow k_e^i - k_e^{i-1}$;
    $v_s^i \leftarrow v_e^{i-1}$;
    $v_l^i \leftarrow v_e^i - v_e^{i-1}$;
  return $\{(k_s^i, k_l^i, v_s^i, v_l^i)\}_{i=0}^{N-1}$;

C. Sort Algorithm

We observe that the Shuffle operation consistently dominates runtime in MapReduce tasks, and this also holds for GPU-accelerated frameworks, where the Shuffle stage can account for more than 80% of total execution time for workloads such as WordCount. Mars relies on a hand-written bitonic sort implementation with $O(n \log^2 n)$ comparison complexity [25].

We invoke `thrust::sort` with the existing application-defined comparator. In our CUDA 12.4 configuration, this selects Thrust's comparison-based parallel merge-sort path [16], while preserving the comparator and Shuffle interface. This replacement changes both the sorting strategy and its implementation; we evaluate the resulting Shuffle-path modernization as a whole.

D. Copy Vectorization

We observe that MapReduce execution on modern GPUs involves substantial on-device data movement. During the Map stage, we find that each thread often copies all or part of its input {inKey, inValue} into intermediate records {interKey, interValue}, and the amount of data moved per thread varies with the task. We further note that byte-wise copying is inefficient on GPUs because memory systems favor accesses to contiguous addresses, and byte-wise copying can exacerbate memory bank conflicts, thereby reducing effective memory bandwidth.

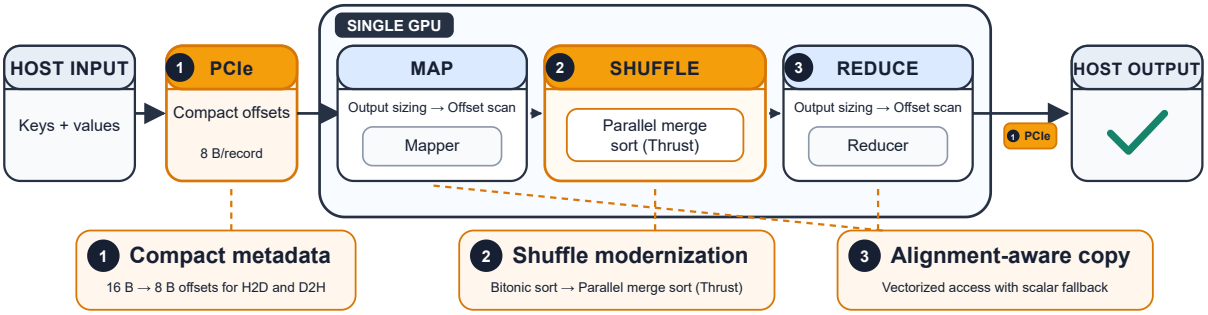


Fig. 1: An overview of HYDRA's workflow.

Algorithm 2: Adaptive Vectorized Copy for MapReduce Kernels

Input: Source pointer src, destination pointer dst, copy size len
Input: Vectorization threshold T
Output: Data copied from src to dst
if $len \geq T$ ***and*** src, dst *are word-aligned* **then**
  Select the widest supported vector type V;
  $n \leftarrow \lfloor len/sizeof(V) \rfloor$;
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $dst[i] \leftarrow src[i]$ using vector loads and stores;
  Copy remaining bytes using byte-wise operations;
else
  for $i \leftarrow 0$ **to** $len-1$ **do**
    $dst[i] \leftarrow src[i]$;

We exploit GPU hardware characteristics by performing vectorized memory copies, reading and writing multiple bytes in a single memory operation to accelerate bulk data movement. Specifically, we aggregate per-thread copy regions whose total size exceeds a predefined threshold and transfer them as coalesced vectorized memory transactions. For example, we pack four 1-byte elements into a 4-byte `int`, eight bytes into an `int2`, and so on. For small or irregularly sized transfers that cannot be handled efficiently by vectorization, we fall back to byte-granular copies, ensuring correctness while maintaining high performance. We formalize this strategy as a per-thread adaptive copy algorithm, shown in Algorithm 2. Copy vectorization substantially reduces on-device data movement time for large transfers. For small transfers, the fallback to byte-wise copying incurs little additional overhead. The main limitation arises when per-thread copy sizes vary irregularly, inducing control-flow divergence that can diminish the benefits of vectorization.

E. Prototype Implementation

We implement a CUDA-based prototype of HYDRA on top of the Mars framework to validate our design. We use Mars' existing abstractions for GPU programming and the MapReduce execution model, providing four system-managed APIs and four user-defined interfaces, thereby reducing the development effort for GPU-based parallel workloads. We extend Mars by compressing the offset metadata transferred over PCIe, replacing the original bitonic sort with Thrust's parallel merge sort, and applying vectorized memory operations instead of byte-wise on-GPU copies. We include one reference use case and eight representative benchmark workloads in the prototype. Users can register input data and implement Map, Compare, and Reduce functions via the provided APIs, and then execute MapReduce jobs through the unified API. Finally, we release the prototype as open source to support reproducibility and future research.

IV. EVALUATION

To assess the effectiveness of our approach, we evaluate HYDRA on real-world benchmarks. Specifically, our evaluation aims to answer the following research questions:

RQ1: Performance. How does HYDRA perform on standard MapReduce workloads?

RQ2: End-to-End Comparison with Mars. How does HYDRA compare with the open-source Mars baseline under the same hardware, software, and workload settings?

RQ3: Ablation Study. What is the performance contribution of each individual HYDRA optimization (*i.e.*, metadata compression, parallel merge sort, and copy vectorization)?

A. Setup

We first describe our experimental setup, including the hardware configuration, evaluation workloads, and baseline system. For all experiments and ablation studies, we perform 10 repetitions and report the mean. We evaluate all workloads on both HYDRA and Mars.

Hardware and Software. We conduct all experiments on a system equipped with an Intel Core i7-12700K CPU, 125 GB of RAM, and a solid-state drive (SSD) as the primary storage device. We perform device-accelerated computations using an NVIDIA GeForce RTX 3060 GPU (12 GB VRAM). Unless otherwise noted, we run experiments on both the CPU and GPU under Ubuntu 25.10 with CUDA 12.4 and NVIDIA driver 580.95.05.

Datasets. Our selection of real-world CUDA applications follows three principles. First, the workloads should be widely used and open source to ensure reproducibility and enable

TABLE I: Execution time breakdown of HYDRA (in ms). N/A: phase not executed.

Workload	PCIe	Map	Shuffle	Reduce	Total
SS	140.36	2704.51	354.92	N/A	3199.83
SM	5.10	6771.58	N/A	N/A	6776.70
MM	87.70	468.59	N/A	N/A	556.30
II	7.59	265.42	386.09	N/A	659.12
PVC	43.67	87.26	900.09	19.11	2126.37
PVR	21.34	19.29	162.16	N/A	202.80
WC	2.41	427.79	1566.05	N/A	1996.27
KM	15.33	7.20	41.15	135.76	205.11

root-cause analysis. Second, they should exhibit diverse computation and communication characteristics, including varying computation-to-data-movement ratios and both map-only and map-shuffle execution patterns. Third, the workloads should cover diverse data types, record sizes, and access patterns, allowing us to evaluate the impact of our optimizations under different key/value layouts and workload intensities.

Based on these principles, we select eight representative MapReduce workloads commonly used in web and data analytics and also implemented in Mars. These workloads include: 1) String Match (SM): Performs exact string matching on input lines, producing matches and their positions; map-only workload. 2) Inverted Index (II): Extracts hyperlink positions from HTML files, outputting link-position pairs; map-shuffle workload. 3) Similarity Score (SS): Computes pairwise cosine similarity between document feature vectors, producing scores and document ID pairs; map-shuffle workload. 4) Matrix Multiplication (MM): Multiplies two matrices by computing dot products of rows and columns, producing result entries keyed by (row, col); map-only workload. 5) Page View Count (PVC): Uses two MapReduce passes: the first deduplicates web logs (<URL, IP, Cookie>), and the second counts distinct page views per URL. 6) Page View Rank (PVR): Ranks URLs by access frequency using PVC output and selects the top ten; map-shuffle workload. 7) Word Count (WC): Counts word occurrences by producing (word, 1) pairs; map-shuffle workload. 8) K-means (KM): Iteratively clusters 3D points into 24 groups using MapReduce [26]. In each iteration, the Map phase assigns points to the nearest cluster centers, and the Reduce phase recomputes centroids until convergence.

B. RQ1: Performance

Table I presents a detailed breakdown of execution time for HYDRA across eight benchmark workloads, including the time spent in each stage and the overall runtime. We observe that the contribution of each stage varies significantly, reflecting the workloads' diverse computation and communication characteristics.

For compute-intensive workloads such as SM and SS, we find that the Map phase dominates the total execution time, accounting for the majority of end-to-end latency. In contrast, for WC and PVC, we identify substantial overhead in the Shuffle phase, indicating that intermediate data aggregation and shuffling act as primary bottlenecks. For lightweight workloads, such as PVR and KM, we observe that execution time is more evenly distributed across stages, with relatively short Map and PCIe phases.

TABLE II: Mars baseline execution time (ms). N/A: phase not executed. Averaged over 10 runs.

Workload	PCIe	Map	Shuffle	Reduce	Total
SS	174.18	2514.31	1648.15	N/A	4336.67
SM	6.26	6738.93	N/A	N/A	6745.20
MM	104.62	608.10	N/A	N/A	712.73
II	9.34	266.91	3731.36	N/A	4007.62
PVC	55.10	81.95	4419.03	21.49	9189.06
PVR	31.70	18.99	486.15	N/A	536.86
WC	3.12	504.57	18266.67	N/A	18774.37
KM	17.55	9.82	102.87	140.33	280.41

TABLE III: HYDRA with all optimizations: execution time (Opt, ms) and speedup vs. Mars (Table II). ↑/↓: speedup $>1.10\times$/$<0.95\times$; N/A: phase not executed.

Wkld	PCIe		Map		Shuffle		Reduce		Total	
	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup
SS	140.36	↑1.24×	2704.51	↓0.93×	354.92	↑4.64×	N/A	N/A	3199.83	↑1.36×
SM	5.10	↑1.23×	6771.58	1.00×	N/A	N/A	N/A	N/A	6776.70	1.00×
MM	87.70	↑1.19×	468.59	↑1.30×	N/A	N/A	N/A	N/A	556.30	↑1.28×
II	7.59	↑1.23×	265.42	1.01×	386.09	↑9.66×	N/A	N/A	659.12	↑6.08×
PVC	43.67	↑1.26×	87.26	↓0.94×	900.09	↑4.91×	19.11	↑1.12×	2126.37	↑4.32×
PVR	21.34	↑1.49×	19.29	0.98×	162.16	↑3.00×	N/A	N/A	202.80	↑2.65×
WC	2.41	↑1.29×	427.79	↑1.18×	1566.05	↑11.66×	N/A	N/A	1996.27	↑9.40×
KM	15.33	↑1.14×	7.20	↑1.36×	41.15	↑2.50×	135.76	1.03×	205.11	↑1.37×

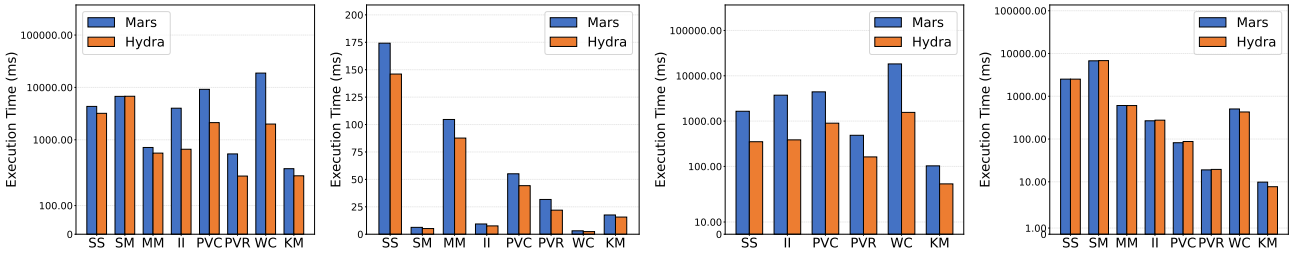
Overall, we demonstrate that HYDRA effectively supports a wide range of MapReduce workloads with distinct performance profiles, and we highlight that optimization opportunities are strongly workload-dependent.

C. RQ2: End-to-End Comparison with Mars

We implement three optimizations (metadata compression, parallel merge sort, and copy vectorization) on top of Mars, and we evaluate all eight workloads under identical conditions. We use Mars because HYDRA directly extends its execution engine, preserves its programming interface, and runs the same workloads under the same hardware and software settings. This controlled comparison does not imply that Mars represents all current GPU data-processing systems. We measure and report the mean execution times for the PCIe Transfer, Map, Shuffle, and Reduce phases, as well as total execution time. For K-means (KM), which performs multiple MapReduce iterations, we average per-phase times across iterations and report total execution time averaged over all iterations.

We summarize our results in Table III and Figure 2a. We observe that, except for String Match (SM), all workloads achieve speedups to varying degrees, with several exhibiting substantial performance improvements.

For total execution time, we find that measurement noise is negligible due to the large execution times; therefore, we treat speedups within $1.01\times$ as noise. We observe that all workloads except SM benefit from our optimizations, with MM achieving the smallest speedup ($1.28\times$) and WC achieving the largest ($9.40\times$). For the PCIe Transfer phase, we observe speedups for all workloads, ranging from $1.14\times$ (KM) to $1.49\times$ (PVR). We attribute this variation to differences in the relative sizes of key, value, and offset fields; by halving offset metadata, we consistently reduce PCIe transfer overhead. For the Map phase,



(a) Total: All optimizations. (b) PCIe: Compression. (c) Shuffle: Sort Algorithm. (d) Map: Copy Vectorization.

Fig. 2: Execution time comparison. Mars (blue) is the baseline; HYDRA (orange) applies the specified optimization(s).

TABLE IV: Ablation: Mars + metadata compression (offset: 16→8 bytes). Opt: execution time (ms); Speedup vs. Mars (Table II). ↑/↓: >1.10×/<0.95×; N/A: not executed.

Wkld	PCIe		Map		Shuffle		Reduce		Total	
	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup
SS	146.04	↑1.19×	2704.94	↓0.93×	1650.57	1.00×	N/A	N/A	4501.58	0.96×
SM	5.17	↑1.21×	6820.42	0.99×	N/A	N/A	N/A	N/A	6825.60	0.99×
MM	87.68	↑1.19×	470.97	↑1.29×	N/A	N/A	N/A	N/A	558.66	↑1.28×
II	7.60	↑1.23×	264.53	1.01×	3731.17	1.00×	N/A	N/A	4003.32	1.00×
PVC	44.27	↑1.24×	80.88	1.01×	4417.73	1.00×	20.81	1.03×	9152.21	1.00×
PVR	21.94	↑1.44×	18.89	1.01×	486.17	1.00×	N/A	N/A	527.03	1.02×
WC	2.41	↑1.29×	504.09	1.00×	18250.01	1.00×	N/A	N/A	18756.53	1.00×
KM	15.65	↑1.12×	9.40	1.04×	102.84	1.00×	136.69	1.03×	274.27	1.02×

TABLE V: Ablation: Mars + Thrust-backed parallel merge sort. Opt: execution time (ms); Speedup vs. Mars (Table II). ↑/↓: >1.10×/<0.95×; N/A: not executed.

Wkld	PCIe		Map		Shuffle		Reduce		Total	
	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup
SS	173.71	1.00×	2513.97	1.00×	350.58	↑4.70×	N/A	N/A	3038.28	↑1.43×
SM	6.21	1.01×	6771.47	1.00×	N/A	N/A	N/A	N/A	6777.69	1.00×
MM	105.63	0.99×	613.06	0.99×	N/A	N/A	N/A	N/A	718.70	0.99×
II	9.36	1.00×	273.03	0.98×	385.33	↑9.68×	N/A	N/A	667.74	↑6.00×
PVC	55.16	1.00×	81.66	1.00×	899.60	↑4.91×	21.46	1.00×	2149.66	↑4.27×
PVR	31.69	1.00×	19.35	0.98×	162.04	↑3.00×	N/A	N/A	213.10	↑2.52×
WC	3.13	1.00×	504.17	1.00×	1556.36	↑11.74×	N/A	N/A	2063.68	↑9.10×
KM	17.41	1.01×	9.74	1.01×	41.15	↑2.50×	134.82	1.04×	209.92	↑1.34×

we find that performance varies across workloads. We observe modest slowdowns for SS and PVC (0.93× and 0.94×), while MM, WC, and KM achieve noticeable speedups. We attribute this behavior to changes in memory-access patterns induced by compression: we find that workloads with high computational intensity benefit from our optimizations, while others do not. We further investigate these effects in the ablation study. For the Shuffle phase, we observe substantial speedups across all workloads that execute this phase, ranging from 2.50× (KM) to 11.66× (WC). This improvement reflects the combined effects of the sorting strategy and Thrust's engineered implementation; the current experiment does not isolate these factors. For the Reduce phase, we execute the Reduce stage for PVC and KM, and we find that both benefit modestly from vectorized memory copies. We demonstrate that combining metadata compression, parallel merge sort, and copy vectorization significantly reduces end-to-end execution time for GPU-accelerated MapReduce workloads.

D. RQ3: Ablation Study

To isolate the contribution of each optimization to per-stage performance, we conduct an ablation study on metadata compression, parallel merge sort, and copy vectorization. Unless otherwise stated, each configuration is executed ten times, and we report mean execution times for all stages.

We enable metadata compression in Mars and compare the resulting performance with the baseline. We report the observed speedups in Table IV and Figure 2b. The results show that metadata compression accelerates PCIe transfer for all workloads to varying degrees, achieving the smallest speedup for KM (1.12×) and the largest for PVR (1.44×). We also observe a Map-stage speedup of 1.29× for MM, whereas SS experiences a slowdown (0.93×). Overall, we achieve a significant speedup of 1.28× in total execution time for MM, while execution times for the remaining workloads remain largely unchanged.

We analyze the effect of metadata compression on PCIe transfer and Map-stage performance and note that workloads differ in the proportions of the total data volume occupied by key, value, and offset fields. For example, in PVR, each 4-byte key and 4-byte value is associated with four 4-byte offsets, so the offset field accounts for 2/3 of the total data volume. We reduce the offset field size from 16 bytes to 8 bytes, which yields the largest PCIe transfer speedup for PVR. By contrast, in KM, the offset fields occupy only 16/44 of the total data volume; we reduce the offset metadata from 16 bytes to 8 bytes, producing a smaller effect on PCIe transfer and explaining the smaller observed speedup.

We further analyze the effect on the Map stage and observe that metadata compression allows the first offset value to be stored in registers rather than fetched from global memory. For compute-intensive workloads such as MM, this significantly reduces memory-access costs and leads to a clear Map-stage speedup. In contrast, for low-intensity workloads such as SS, the additional control divergence outweighs the memory-access savings, resulting in a slowdown. For the other workloads, we observe no significant change in Map-stage performance. We enable Thrust's parallel merge-sort path in Mars and compare its performance against the original bitonic sort baseline, with results summarized in Table V and Figure 2c. The results show that this change substantially accelerates all workloads featuring a Shuffle stage. Specifically, we measure Shuffle-stage speedups ranging from 2.50 × (KM) to 11.74 × (WC). Crucially, we find that the optimization introduces no

TABLE VI: Ablation: Mars + copy vectorization (4-byte operations when size >8 bytes). Opt: execution time (ms); Speedup vs. Mars (Table II). ↑/↓: >1.10×/<0.95×; N/A: not executed.

Wkld	PCIe		Map		Shuffle		Reduce		Total	
	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup	Opt	Spdup
SS	174.77	1.00×	2513.55	1.00×	1647.86	1.00×	N/A	N/A	4336.21	1.00×
SM	6.18	1.01×	6816.61	0.99×	N/A	N/A	N/A	N/A	6822.80	0.99×
MM	104.83	1.00×	606.11	1.00×	N/A	N/A	N/A	N/A	710.95	1.00×
II	9.37	1.00×	276.09	0.97×	3730.78	1.00×	N/A	N/A	4016.26	1.00×
PVC	55.42	0.99×	87.47	↓0.94×	4419.82	1.00×	19.98	↑1.08×	9201.18	1.00×
PVR	32.20	0.98×	19.48	0.97×	486.06	1.00×	N/A	N/A	537.76	1.00×
WC	3.16	0.99×	429.25	↑1.18×	18256.48	1.00×	N/A	N/A	18688.91	1.00×
KM	17.46	1.01×	7.67	↑1.28×	102.95	1.00×	137.42	1.02×	275.68	1.02×

measurable overhead in other pipeline stages; consequently, these workloads achieve end-to-end speedups ranging from 1.34 × (KM) to 9.10 × (WC).

Workloads with longer initial Shuffle times benefit most: WC, which incurs the highest baseline Shuffle cost, achieves the largest speedup, whereas KM, whose Shuffle phase is initially the shortest, shows the smallest improvement. This comparison changes both the sorting strategy and its implementation. Therefore, the measured speedup should be interpreted as the benefit of replacing Mars' legacy Shuffle implementation with a maintained GPU primitive, rather than as an algorithm-only comparison. A controlled comparison between equally optimized implementations is left for future work.

We enable copy vectorization in Mars and evaluate it against the baseline implementation. We summarize the results in Table VI and Figure 2d. The results show that the effects of copy vectorization vary across workloads and execution stages. For PVC, we observe a Map-stage slowdown (speedup ratio: 0.94×), while the Reduce stage achieves a speedup of 1.08×. For WC and KM, we observe Map-stage speedups of 1.18× and 1.28×, respectively. For the remaining workloads and stages, we observe no notable performance changes.

We analyze these results based on key and value sizes relative to the vectorization threshold.

For KM, we note that both the key and value are 24 bytes, exceeding the 8-byte threshold for vectorized memory copying. We replace byte-wise copies with six 4-byte vector copies for each key and value, allowing KM to fully exploit vectorized memory operations. In this case, the performance gains from vectorized copying outweigh the additional control overhead, yielding a clear Map-stage speedup.

For WC, we observe that keys are variable-length words with an average size close to the vectorization threshold (approximately 8–15 bytes). We vectorize a subset of key copies, while values remain 4 bytes in size and are copied byte-wise. Although vectorization introduces additional control logic, the benefits from key copying dominate, resulting in an overall reduction in Map-stage execution time.

For PVC, we attribute the Map-stage slowdown to mixed copy conditions across pipeline stages. In the first Map stage (key = 16 bytes, value = 8 bytes), we successfully amortize the branching overhead of copy vectorization through accelerated key copying. In the second stage (key = 16 bytes, value = 4 bytes), the value size falls below the vectorization threshold, and the per-record control overhead becomes more pronounced. In this case, the key-copy acceleration is insufficient to offset the added overhead, leading to a net Map-stage slowdown. We observe that the Reduce stage follows the same access pattern as the first Map stage and therefore benefits from vectorized key copies, explaining the observed speedup.

Overall, we find that copy vectorization is effective when both key and value sizes exceed the vectorization threshold. When only a subset of fields can be vectorized, the additional control overhead can partially or fully offset the performance benefits.

V. Related Work

Efforts to accelerate MapReduce fall into three categories.

Hardware-aware acceleration exploits specialized processors. Hong et al. [23] enhance Spark with GPU in-memory caching and MPI for interactive visualization. Algahtani et al. [11] [12] use CPU SIMD and multi-GPU clusters in Hadoop to parallelize inductive logic learning, achieving a speedup of up to 161×. Seliem et al. [27] deploy a MapReduce-based sequence aligner on FPGAs, while Zhong et al. [28] optimize MapReduce for the Sunway many-core architecture via local memory and dynamic task partitioning.

Shuffle and communication optimization targets data movement bottlenecks. Wu et al. [29] propose Shadow, which duplicates map tasks across racks to balance cross-rack traffic, cutting job time by 26%. Tyryshkina [30] redesigns distributed joins in Spark, accelerating 7 TB merges by 47%. Addisie et al. [31] introduce CASM, collaborative accelerators that reduce off-chip traffic by 4× on scale-up machines.

Hybrid acceleration and integration with existing ecosystems. Algorithmic and workload-aware refinement reduces computational redundancy. Bei et al. [32] present OSC, an online tuner that auto-configures Hadoop parameters, improving performance by up to 120%. Han et al. [33] propose SlimML, which trains ML models only on "critical" data, yielding a 3.6× speedup with minimal accuracy loss. Others apply domain-specific optimizations: Liu et al. [34] and Li [35] accelerate microgrid simulation and image segmentation via pipeline tuning; Maillo et al. [36] and Wangming [37] integrate approximate or association-mining algorithms for scalable analytics.

Hydra instead focuses on modernizing the execution path of a single-GPU MapReduce engine while preserving its programming interface.

VI. Conclusion

We presented Hydra, a modernization of the Mars single-GPU, in-memory MapReduce execution engine. Hydra improves the efficiency of GPU-based MapReduce workloads through a set of complementary optimizations. Specifically, it reduces CPU–GPU data-transfer overhead through metadata compression, accelerates GPU-side grouping by replacing bitonic sort with Thrust's parallel merge sort, and reduces

intra-GPU data-movement overhead using vectorized memory copy operations. We implemented a software prototype and evaluated it on eight representative workloads. HYDRA achieves a geometric mean speedup of 2.51× and a maximum speedup of 9.40× over Mars, with the effects of metadata compression and vectorized copying varying across workloads. Overall, the results demonstrate the benefit of modernizing the execution path of a legacy single-GPU MapReduce engine.

REFERENCES

[1] A. Chennupati, B. Prahas, B. A. Ghali, and M. Venugopalan, "Scalable Web Crawling: Harnessing the Power of Hadoop MapReduce in a Distributed Framework," in *2023 Global Conference on Information Technologies and Communications (GCITC)*, Dec. 2023, pp. 1–5.

[2] S. S. Vernekar and A. Buchade, "MapReduce based log file analysis for system threats and problem identification," in *2013 3rd IEEE International Advance Computing Conference (IACC)*, Feb. 2013, pp. 831–835.

[3] T. Jingwen, C. Weimin, Z. Min, P. Kaikang, L. Chaohao, and F. Kai, "Research on log data analysis technology based on improved Hadoop," in *2022 IEEE 2nd International Conference on Data Science and Computer Application (ICDSCA)*, Oct. 2022, pp. 271–274.

[4] A. Bhatt, A. Uniyal, D. Jyala, S. Mittal, P. Tiwari, and D. Singh, "Resume Analyzer based on MapReduce and Machine Learning," in *2024 IEEE International Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI)*, vol. 2, Mar. 2024, pp. 1–5.

[5] H. Ouhakki and A. Elmoufidi, "An implementation of GPU accelerated mapreduce: Using hadoop with openCL for breast cancer detection and compute-intensive jobs," *International Journal of Information Technology*, vol. 17, no. 5, pp. 2605–2612, Jun. 2025.

[6] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Toronto Ontario Canada: ACM, Oct. 2008, pp. 260–269.

[7] M. Elteir, H. Lin, W.-c. Feng, and T. Scogland, "StreamMR: An Optimized MapReduce Framework for AMD GPUs," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, Dec. 2011, pp. 364–371.

[8] Z. Qiao, S. Liang, H. Jiang, and S. Fu, "MR-Graph: A Customizable GPU MapReduce," in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, Nov. 2015, pp. 417–422.

[9] Y. Guo, W. Liu, B. Gong, G. Voss, and W. Muller-Wittig, "GCMR: A GPU Cluster-Based MapReduce Framework for Large-Scale Data Processing," in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov. 2013, pp. 580–586.

[10] Z. Tang, L. Du, X. Zhang, L. Yang, and K. Li, "AEML: An Acceleration Engine for Multi-GPU Load-Balancing in Distributed Heterogeneous Environment," *IEEE Transactions on Computers*, vol. 71, no. 6, pp. 1344–1357, Jun. 2022.

[11] E. Algahtani, "MP-HTHEDL: A Massively Parallel Hypothesis Evaluation Engine in Description Logic," *IEEE Access*, vol. 12, pp. 89 113–89 123, 2024.

[12] ——, "MP-SPILDL: A Massively Parallel Inductive Logic Learner in Description Logic," *IEEE Access*, vol. 12, pp. 130 884–130 895, 2024.

[13] S. Hong, W. Choi, and W.-K. Jeong, "GPU in-Memory Processing Using Spark for Iterative Computation," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 31–41.

[14] M. F. Lehnen, L. M. Schnorr, and P. O. A. Navaux, "Can GPUs help scaling traditional Apache Spark workloads?" in *2025 IEEE/SBC 37th International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct. 2025, pp. 69–76.

[15] M. Xin and H. Li, "An Implementation of GPU Accelerated MapReduce: Using Hadoop with OpenCL for Data- and Compute-Intensive Jobs," in *2012 International Joint Conference on Service Sciences*, May 2012, pp. 6–11.

[16] "NVIDIA/thrust," NVIDIA Corporation, Jan. 2026.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[18] T. Chowdhury, C. Bhattacharya, S. Chowdhury, M. K. Nath, and M. De, "GPU Accelerated MapReduce-Based Distributed Framework for Knowledge Extraction from Large Uncertain Data," *SN Computer Science*, vol. 5, no. 8, p. 1117, Nov. 2024.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010, pp. 1–10.

[20] C. Liu and S. Zhou, "Local and Global Optimization of MapReduce Program Model," in *2011 IEEE World Congress on Services*, Jul. 2011, pp. 257–264.

[21] Y. Zheng, Y. Xu, H. Meng, X. Ye, L. Fan, F. Miao, and D. Fan, "Optimizing mapreduce with low memory requirements for shared-memory systems," in *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, Jun. 2014, pp. 1–6.

[22] E. Selin and H. Viktor, "XRT: Programming-Language Independent MapReduce on Shared-Memory Systems," in *2018 IEEE International Congress on Big Data (BigData Congress)*, Jul. 2018, pp. 182–189.

[23] S. Hong, J. Choi, and W.-K. Jeong, "Distributed Interactive Visualization Using GPU-Optimized Spark," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 9, pp. 3670–3684, Sep. 2021.

[24] "CUDA Programming Guide," https://docs.nvidia.com/cuda/cuda-programming-guide/.

[25] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: Association for Computing Machinery, Jun. 2006, pp. 325–336.

[26] W. Zhao, H. Ma, and Q. He, "Parallel K-Means Clustering Based on MapReduce," in *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds. Berlin, Heidelberg: Springer, 2009, pp. 674–679.

[27] A. G. Seliem, H. F. A. Hamed, and W. Abouelwafa, "MapReduce Model Using FPGA Acceleration for Chromosome Y Sequence Mapping," *IEEE Access*, vol. 9, pp. 83 402–83 409, 2021.

[28] X. Zhong, M. Li, H. Yang, Y. Liu, and D. Qian, "swMR: A Framework for Accelerating MapReduce Applications on Sunway Taihulight," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 1020–1030, Apr. 2021.

[29] S. Wu, H. Chen, H. Jin, and S. Ibrahim, "Shadow: Exploiting the Power of Choice for Efficient Shuffling in MapReduce," *IEEE Transactions on Big Data*, vol. 8, no. 1, pp. 253–267, Feb. 2022.

[30] Y. Tyryshkina, "Accelerating Join of Distributed Datasets by a Given Criterion," in *2022 Moscow Workshop on Electronic and Networking Technologies (MWENT)*, Jun. 2022, pp. 1–3.

[31] A. Addisie and V. Bertacco, "Collaborative Accelerators for Streamlining MapReduce on Scale-up Machines With Incremental Data Aggregation," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1233–1247, Aug. 2020.

[32] Z. Bei, N. S. Kim, K. HWang, and Z. Yu, "OSC: An Online Self-Configuring Big Data Framework for Optimization of QoS," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 809–823, Apr. 2022.

[33] R. Han, C. H. Liu, S. Li, L. Y. Chen, G. Wang, J. Tang, and J. Ye, "SlimML: Removing Non-Critical Input Data in Large-Scale Iterative Machine Learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 5, pp. 2223–2236, May 2021.

[34] J. Liu, Z. Wu, Z. Huang, S. Pu, H. Chen, and J. Gao, "Mircogrid Voltage Risk Evaluation Using Distributed Parallel Monte-Carlo Simulation," in *2020 IEEE Sustainable Power and Energy Conference (iSPEC)*, Nov. 2020, pp. 325–330.

[35] M. Li, L. Meng, J. Wang, Y. Jin, B. Hu, and Y. Chen, "Application and Performance Optimization of MapReduce Model in Image Segmentation," *IEEE Access*, vol. 8, pp. 31 835–31 844, 2020.

[36] J. Maillo, S. García, J. Luengo, F. Herrera, and I. Triguero, "Fast and Scalable Approaches to Accelerate the Fuzzy k-Nearest Neighbors Classifier for Big Data," *IEEE Transactions on Fuzzy Systems*, vol. 28, no. 5, pp. 874–886, May 2020.

[37] C. Wangming, "Analysis and Application: Sports Education Information Management under the Background of Big data," in *2021 13th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, Jan. 2021, pp. 543–546.