High-Throughput Chunk-Based GPU Bitmap Compression for Dense Retrieval Workloads

José Pedro de Santana Neto[1], Simone Dominico[1], Eduardo Henrique Monteiro Pena[2], Eduardo Cunha de Almeida[1], and Marco Antonio Zanata Alves[1]

[1] Federal University of Paraná, Brazil
jose.santana@ufpr.br, {sdominico, eduardo, mazalves}@inf.ufpr.br
[2] Federal University of Technology - Paraná, Brazil
eduardopena@utfpr.edu.br

Abstract. Compressed bitmap indexes are a means of executing queries efficiently while reducing memory usage. However, most existing bitmap formats are designed for CPUs and exhibit irregular control flow and memory access patterns that limit performance on GPUs. This paper presents HiPES, a GPU-native, chunk-based compressed bitmap scheme. Experimental results show that HiPES can achieve an 8× increase in compression throughput relative to GPU-WAH in dense workloads, while maintaining competitive compression ratios. In sparse workloads, the efficiency degrades due to internal fragmentation, associated with the fixed-size chunks layout. When the chunk payload bit-width (w) is aligned with shared memory, the throughput is maximized. Additionally, for values of w around 16, we reach a balance in terms of efficiency in which both metadata overhead and internal fragmentation are minimized. This work establishes a basis for GPU-resident computational frameworks, allowing for the elimination of the overhead associated with decompression in high-scale filtering, Graph Neural Networks (GNN) pre-processing, and in-situ scientific analytics.

Keywords: High-throughput algorithms · Data-intensive workloads · Dense retrieval · GPU-optimized compression · In-situ data processing

1 Introduction

Bitmap indexes provide a bridge between hardware bitwise operations and higher level of abstraction for data filtering. While historically rooted in database systems, these techniques have become vital for data-intensive workloads that require large scale intersections and aggregations [22,16]. However, for large datasets, the storage overhead of a bitmap index is proportional to both the number of rows and the number of distinct values [5]. To overcome this limitation and preserve data processing performance, bitmap indexes can be stored compressed to reduce I/O and memory footprint. Nonetheless, performance gains depend on the balance between data movement reduction and the computational cost of compression and decompression [23].

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

The current trends toward GPU acceleration have been able to take advantage of the massive parallelism needed for data-intensive applications. This enables simultaneous execution of logical operations across thousands of threads [22,28,20]. However, existing bitmap compression techniques, such as the Word-Aligned Hybrid (WAH) [29], Enhanced Word-Aligned Hybrid (EWAH), Concise [7] and Roaring Bitmap [16,17], were originally designed for traditional scalar processing on CPU execution. As a result, these techniques make use of a high number of control dependencies and irregular access patterns that can hinder GPU parallelism [28,24]. Thus, the development of bitmap compression algorithms that achieve optimal GPU resource utilization is still an open problem [28,21].

This paper presents the Hierarchical Performant and Efficient Set (HiPES) bitmap scheme. It encompasses a data structure and compression algorithm optimized for large-scale parallel processing, coalesced memory access, low thread divergence, and on-chip memory usage. We analyze how the chunk payload bitwidth (w) affects the HiPES scheme, addressing the following questions: (1) How will the alignment of w with the GPU memory hierarchy affect the compression throughput? (2) How will the w affect the compression efficiency for different data workloads? We compare it to GPU-WAH [22,28] as the standard GPU-ported scheme. Other CPU-based bitmap format schemes use data-dependent control flow and non-uniform layouts that are fundamentally different from those required to be efficient on a GPU, which is beyond our scope. Apart from traditional database indexing, HiPES targets a growing class of GPU-resident data processing workloads. Examples include metadata filtering in large-scale ensembles of simulations, Graph Neural Networks (GNN) pre-processing using set intersection, candidate pruning in retrieval-augmented generation (RAG) and vector search, in-situ data analytics to reduce data movement, and data quality workloads that rely heavily on set operations [27,25,30], among others.

2 Related Work

Bitmap compression schemes have evolved primarily for CPU architectures. Early methods such as WAH [29], EWAH [13], PLWAH [2], Concise [7], and Roaring [16,4,17] rely on variable-length encoding and hybrid containers designed for scalar execution. GPU adaptations include GPU-WAH and GPU-PLWAH [1,2]. In GPU-WAH, the compression is accelerated through the steps of word classification, prefix-sum scans, and parallel writes. GPU-PLWAH improves compression ratios by encoding dissenting bit positions into preceding fill words, though this often requires reconversion to WAH for efficient querying. While GPU-PLWAH can have improvements in terms of compression ratios, its performance degrades compared to GPU-WAH [21,2]. While both outperform CPU implementations, they remain limited in their ability to fully utilize the GPU hardware architecture.

Subsequent work focused on decompression and query execution rather than compression [22,28,21,20,26], typically offloading compression to the CPU. This

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

highlights a natural limitation because there is a dependence on additional operations to establish queries. Also, the mentioned studies point to the lack of research about optimized compression schemes for GPUs [28,22]. Hybrid CPU/GPU approaches reduce some bottlenecks but introduce data transfer overhead [6,31,23]. Specialized GPU-native frameworks like Falcon [18] and BLEST [10] demonstrate the benefits of this paradigm in other domains. Yet, despite recent GPU adaptations of Roaring formats [9], native GPU bitmap compression optimized for shared-memory (SMEM), coalesced access, and SIMT execution remains an open problem [28,21]. A summary of the main characteristics of the above-mentioned schemes is available in the Artifacts.

3 HiPES Bitmap Scheme

3.1 Data Structure

The data structure of HiPES is based on sets of unique 32-bit integers. These integers can represent either identifiers or keys of elements in a dataset, such as record IDs or encoded attribute values. HiPES partitions the 2^{32} address space into disjoint fixed-size blocks, hereafter referred to as chunks, each encompassing $2^{16} = 65{,}536$ potential elements. Each 32-bit integer x is decomposed into a prefix and a suffix. The prefix is the chunk identifier (16 Most Significant Bits, MSBs). The suffix encodes the local bit position within the chunk (16 Least Significant Bits, LSBs). With the HiPES scheme, we have the opportunity for compression, since there are no empty chunks being stored, along with GPU alignment, through regular and uniform memory layout. The way that chunks are organized is the core of HiPES, since it allows us to eliminate the irregularities that lead to GPUs underperforming while maintaining the ability to perform set operations directly on the compressed data. The choice of 16-bit boundary is dictated by architectural constraints. It partitions the 32-bit address space at its natural midpoint, yielding bitmaps of 8,192 bytes that align with a single GPU memory hierarchy. This size fits within a single SMEM allocation per thread block. It allows the use of intra-chunk reductions and warp-level primitives for high-performance inter-thread coordination. HiPES utilizes a bijective transformation that provides a lossless property and preserves the unique (one-to-one) mapping between an input set S and its compressed form. This transformation is reversible and guarantees that the reconstructed set S' is identical to S, eliminating false positives or loss of information. Each element x is associated with a chunk identifier c (the 16 MSBs) and a bit position p (the 16 LSBs). The reconstruction is given by $x = (c \ll 16) \vee p$. When the number of rows exceeds 2^{32}, the payload can be extended to 64-bit row identifiers. However, this still allows for the same block size and coalescing of memory access. This property also provides the capability of performing bitwise operations directly on the compressed data. Furthermore, the structure of the compressed data is aligned with the SIMT model.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

3.2 Compression Algorithm

HiPES achieves compression by discarding empty chunks and by dividing each 32-bit integer into fixed-size blocks. The same hierarchical structure also enables direct execution of bitwise operations on compressed data. The entire process is organized as a sequence of five passes. Each pass produces intermediate data that feeds the next stage. This allows the entire process to run efficiently on the GPU without host-to-device transfers. Algorithm 1 shows the overall flow. It consists of five GPU-parallel stages, labeled as Pass 0 to Pass 4. The process begins with the identification of chunks that are non-empty, maps them to dense consecutive indices, tags each element, groups elements belonging to the same chunk, and finally builds the fixed-size bitmaps using SMEM. For illustrative purposes, consider the following running examples: *Set A* and *Set B*, in which *Set A* is composed of 13 elements, clustered at the start with two distant outliers, represented as *Set A* $= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 131{,}075, 2{,}228{,}227\}$. *Set B* is composed of 12 elements, similar to A, but spans an additional chunk, represented as *Set B* $= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 65{,}536, 131{,}075, 2{,}228{,}227\}$.

Pass 0 identifies which of the 2^{16} possible chunks actually contain elements. This step avoids memory allocation or computation on empty regions. Traditional run-length encoding (RLE)-based approaches identify non-empty segments implicitly through a full $O(n \log n)$ sort of the input. HiPES, therefore, replaces this with a single parallel scan that sets one bit per element in the `GCB` using atomic operations, reducing the identification step to $O(n)$ time with no inter-element ordering dependency. Each chunk corresponds to a group of 32-bit integers that share the same 16 MSB, as explained in Section 3. This pass implements this identification strategy. It is achieved by iterating through the input set S in parallel. For each integer, its 16 most significant bits (MSBs) are extracted to serve as a chunk identifier. This identifier is used to atomically set a corresponding bit in a 65,536-bit global bitmap. Then, in order to serve as a summary of all active chunks, the output is a compact bitmap named `GCB`. It is a global bitmap composed of 2,048 contiguous 32-bit words, where each bit indicates the presence of a non-empty chunk. Using *Set A*, this pass identifies three non-empty chunks (IDs 0, 2, and 34). Using *Set B*, it identifies four non-empty chunks (IDs 0, 1, 2, and 34).

Within Pass 1, the sparse logical chunk IDs are transformed into a dense and contiguous set of indices. It enables coalesced memory access in all subsequent stages. The process begins with the computation of the total number of non-empty chunks, n_c. This is done by performing a parallel population count and an exclusive scan on the `GCB`. Then, two mapping arrays (`chunk_to_index` and `index_to_chunk`) are constructed. The `chunk_to_index` recovers an incremental index by using the chunk identifier. The `index_to_chunk` reconstructs the chunk header (the 16-bit identifier representing the MSB of all integers within that chunk), based on the index. These maps are crucial to minimize warp divergences. In the last pass, the `index_to_chunk` map plays a meaningful role in the final data structure creation. The auxiliary function `pop_offset_within_word`$(word, b)$ computes the number of active bits preceding position b within

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

the 32-bit word *word*. It is used to calculate the local offset of a chunk within its word. This allows each active chunk to receive a unique, compact index in the contiguous mapping. For Set A, the sparse logical Chunk IDs 0, 2, 34 would be converted to dense indices 0, 1, 2 and for Set B, the IDs 0, 1, 2, 34 would be converted to indices 0, 1, 2, 3. This mapping allows for contiguous memory allocation and coalesced accesses in subsequent passes.

Pass 2 creates a compact auxiliary structure (`BitEntry`) containing only the information needed for the remaining steps. Thus, it converts a complex sorting problem into a simple, memory-bound transformation that runs in linear time. The `BitEntry` array has the same size as the input set S. Each element of the `BitEntry` array stores a pair of `(chunk_idx, bit_pos)`. The `chunk_idx` is a 32-bit unsigned integer (`uint32_t`) representing the compact chunk index, and `bit_pos` is a 16-bit unsigned integer (`uint16_t`) corresponding to the bit position within that chunk. A critical distinction exists between the `chunk_id` and the indexed value for `chunk_idx`. The `chunk_id` is the original 16-bit identifier extracted from the MSBs of the input integers. Meanwhile, the indexed

Algorithm 1 HiPES Compression Pipeline

Require: Input set S of unique 32-bit integers
Ensure: Compressed HiPES structure stored in GMEM
// **Pass 0: Identify Active Chunks**
$GCB \leftarrow \{0\}$ // Initialize Global Chunk Bitmap
for $val \in S$ **do in parallel**
  atomicOr($GCB[\text{MSB}(val)], \text{bit\_mask}(val)$)
end for
// **Pass 1: Chunk Mapping**
$PopCntPerWord \leftarrow \text{PopCount}(GCB)$ // Per-word population count
$Offsets \leftarrow \text{ExclusiveScan}(PopCntPerWord)$
$n_c \leftarrow \text{PopCount}(GCB)$ // Total non-empty chunks
$chunk\_to\_index, index\_to\_chunk \leftarrow \text{GenerateMaps}(GCB, Offsets)$
// **Pass 2: Element Tagging**
$BitEntry \leftarrow \text{Array}(|S|)$
for $i \leftarrow 0$ to $|S| - 1$ **do in parallel**
  $chunk\_idx \leftarrow chunk\_to\_index[\text{MSB}(S[i])]$
  $BitEntry[i] \leftarrow (chunk\_idx, \text{LSB}(S[i]))$
end for
// **Pass 3: Binning**
$Hist \leftarrow \text{ComputeHistogram}(BitEntry.chunk\_idx)$
$BinOffsets \leftarrow \text{ExclusiveScan}(Hist)$
$BinnedBitEntry \leftarrow \text{Scatter}(BitEntry, BinOffsets)$
// **Pass 4: HiPES Construction**
for $c \leftarrow 0$ to $n_c - 1$ **do in parallel**
  $S\_Bitmap \leftarrow \{0\}$ // Allocate 8KB in SMEM
  $Range \leftarrow (BinOffsets[c], BinOffsets[c+1])$
  for $e \in BinnedBitEntry[Range]$ **do**
    atomicOr($S\_Bitmap[e.bit\_pos \gg 5],\ (1 \ll (e.bit\_pos\ \&\ 31))$)
  end for
  Barrier() // Ensure SMEM consistency
  GlobalWrite($index\_to\_chunk[c], S\_Bitmap$)
end for

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

value `chunk_idx` is the index assigned during Pass 1. The values `chunk_idx` and `bit_pos` correspond, respectively, to 16 MSBs and 16 LSBs of each element in S. In order to apply the lookup table procedure, `chunk_idx` is used in `chunk_to_index`. The LSBs represent the bit position (`bit_pos`) of that chunk. This format enables the execution of a parallel sort and runs with $O(n)$ complexity, instead of traditional sorts with $O(n \log(n))$. This preserves the linear-time complexity, and at the same time, it groups elements efficiently. For *Set A*, this pass produces 13 BitEntry pairs. For instance, values 0-10 receive `chunk_idx`=0 and their respective bit positions, while 131,075 is tagged as (`chunk_idx`=1, `bit_pos`=3) and 2,228,227 as (`chunk_idx`=2, `bit_pos`=3). For *Set B*, it produces 12 BitEntry pairs, tagging values 0-8 with `chunk_idx`=0 and bit positions 0-8, 65,536 with `chunk_idx`=1 and bit position 0, and the two outliers with their respective indices.

The purpose of Pass 3 is to group all elements belonging to the same chunk in contiguous memory, so that a single block of threads can process all data for a given chunk from a contiguous memory segment without cross-block coordination. This is the step that would require an $O(n \log n)$ comparison sort in CPU-oriented schemes. HiPES replaces it with the $O(n)$ count-prefix-scatter pattern, using per-block shared-memory histograms to eliminate the global atomic contention that a direct scatter into global memory would incur. Thus, the binning process groups the tagged elements based on the `chunk_idx` of each `BitEntry`. By employing a parallel "count-prefix-scatter" pattern, this stage is analogous to a single-pass radix sort [8,14]. First, the number of elements per chunk is determined by a histogram. Then, an exclusive scan computes the starting memory offset for each chunk bin. Finally, each `BitEntry` is scattered in parallel to its resulting sorted position in the `BinnedBitEntry` array. For *Set A*, this pass groups the 11 elements of Chunk 0 contiguously, followed by the single element of Chunk 2 and Chunk 34, respectively. For *Set B*, it groups the 9 elements of Chunk 0, the single element of Chunk 1, Chunk 2, and Chunk 34, respectively. This step creates the necessary data locality for the construction in the final pass.

The final pass constructs the actual HiPES bitmap using a chunk-centric parallelization strategy. By using the data locality (Pass 3), each non-empty chunk is assigned to a dedicated block of parallel threads. This block processes the corresponding contiguous slice of the `BinnedBitEntry` array. A fixed-size 8 KB bitmap is allocated in the SMEM. Then, each thread atomically sets the bits corresponding to the `bit_pos` of its assigned elements. The use of SMEM in this step is vital. This avoids high-latency, high-contention atomic operations on global memory. Once all elements have been processed, a synchronization barrier is applied. Then, a single thread performs one contiguous write operation. It then merges the chunk header with the 8 KB bitmap and writes it directly into global memory. The result is the final compressed HiPES data structure. For *Set A*, three thread blocks will be launched to build the full bitmaps for each of the Chunks 0, 2, and 34. Similarly, for *Set B*, four thread blocks will be launched to build the full bitmaps for each of the Chunks 0, 1, 2, and 34.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

3.3 Chunk Payload Bit-Width (w)

HiPES uses a fixed bit-width for the payload of each compressed bitmap block, referred to as w, which is determined by the GPU memory hierarchy. This guarantees consistent control-flow and memory-access behavior, and enables massive parallelism with reduced thread divergence, coalesced memory access, and mitigation of global atomic contention through SMEM operations [24,12]. While it may cause internal fragmentation in individual blocks, the chunk-based design ensures overall performance consistency compared to variable-length encoding or branch conditions in other GPU schemes. This aligns with the memory-bound paradigm [11], minimizing direct accesses to global memory (GMEM). To investigate the impact of chunk granularity, w is used as a design variable determining memory bits allocated to each LSB suffix. Considering a base case of $w = 16$, for $w < 16$ chunk bitmaps will decrease in size, resulting in finer granularity but higher metadata overhead due to an increased number of chunk headers. For $w > 16$, the size of the bitmap will increase (2^w), reducing the total number of chunks and their respective headers.

3.4 Design Implications

This design introduces implications between memory footprint and alignment efficiency. The total memory footprint of the compressed structure is exactly $M_{\text{total}} = n_c \times 8{,}196$ bytes, where n_c is the number of materialized (non-empty) chunks. Peak memory during construction scales linearly as $O(n)$ due to temporary tagging and binning buffers. The theoretical compression ratio can be expressed in two complementary forms. The first is $C_{\text{ratio},t} = 4n/(n_c \times 8{,}196)$, where n is the cardinality of the input set and the factor 4 accounts for the original 32-bit integer size. An equivalent formulation incorporates average internal fragmentation per chunk as $C_{\text{ratio},t} = 32\,(1 - F_{c,\text{avg}}/8{,}192\,)$, where $F_{c,\text{avg}}$ is the average fragmentation (unused bytes) across all materialized chunks. Thus, the theoretical maximum compression ratio is $32\times$ as fragmentation is eliminated ($F_{c,\text{avg}} \to 0$), i.e., when every 32-bit integer is fully packed into the compressed representation. Therefore, the compression is achieved whenever the average number of elements per chunk exceeds ≈2,049 (i.e., local density $= k_c/65{,}536 > 3.13\%$). Below this threshold sparse workloads experience data expansion. The internal fragmentation per chunk is quantified as $F_c = 8{,}192 \times (1 - (k_c/65{,}536))$, where k_c is the number of set bits actually present in that chunk. According to it, sparse distributions incur higher fragmentation, while dense or clustered distributions benefit from the removal of RLE irregularities. To illustrate this behavior, the theoretical maximum compression curve across the two primary data placement regimes (expansion and compression) are presented in Fig. 1.

While HiPES shares some similarities with Roaring bitmaps [16], it is specialized for GPUs, ensuring SIMT hardware utilization, coalesced access and GPU memory hierarchy alignment. While Roaring utilizes hybrid containers to manage varying data densities on CPUs, HiPES currently targets moderate-to-dense datasets through a uniform, fixed-block structure that guarantees predictable

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

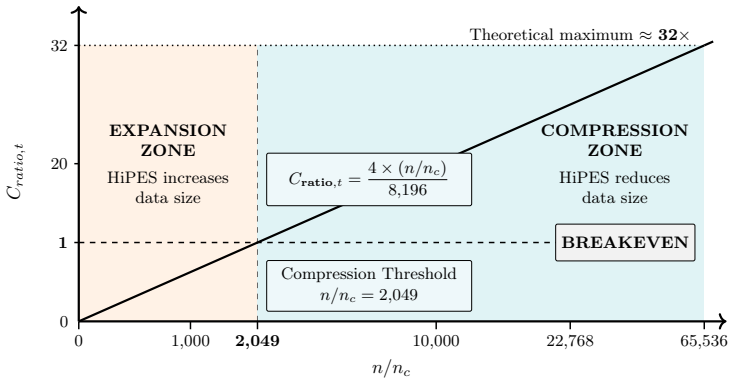


Fig. 1: Schematic behavior of the HiPES theoretical compression ratio ($C_{ratio,t}$) as a function of chunk density (n/n_c).

control flow across SIMT threads. Recent developments have been made to adapt Roaring bitmap hybrid container formats in NVIDIA's cuCollections library [9]. Yet, they continue to face performance penalties from control-flow divergence when managing varied container types. Adaptation of HiPES must be developed to address the challenges for sparse data removing overheads in performance[19] and are left for future research. HiPES thus shows that hardware-aligned fixed blocks can achieve high-throughput without the branch divergence of container-based hybrid formats.

3.5 CUDA Mapping and Implementation Details

The HiPES bitmap scheme is mapped into a sequence of CUDA kernel launches and NVIDIA Thrust library function calls. Additionally, the algorithm can be readily ported to other GPU platforms. Since GPU atomics require `uint32_t`, all bitmap words use that type [24]. A fallback to global-memory atomics activates when per-chunk SMEM requirements exceed hardware capacity. Details of the full CUDA implementation can be found in the Artifacts.

4 Experimental Setup

We focus our experimental comparison on GPU-WAH, as it represents a standard GPU-ported bitmap compression scheme [22,28]. While also ported to GPU, GPU-PLWAH [2], was excluded because it achieves lower compression throughput than GPU-WAH and requires an additional conversion step back for bitwise processing. Other schemes[7,9,16,4], rely on serial, data-dependent logic and hierarchical metadata, which may hinder efficient GPU parallelization. GPU-native frameworks specialized in different domains (i.e., Falcon [18]) were likewise not considered, as their application-level semantics differ from integer-based bitmap indexing.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

4.1 GPU-WAH Implementation

The original GPU-WAH algorithm [22,28,1] uses the parallelism of SIMT to perform various operations on a pre-materialized raw bitmap. The extension phase takes each raw 32-bit word and converts it into a 31-bit WAH word. It crosses word boundaries with 64-bit temporary values and applies masks appropriately, using constants `BM31 = 0x7FFFFFFF` and `BMMSB = 0x80000000`. The phase of boundary detection identifies both where the most significant bit has been set for uniform fill words (all-0s or all-1s), and the block boundaries available when a literal word exists or adjacent fill words change type. In the compression phase, run lengths of identical fill blocks are encoded and generate the final compressed bitmap. In contrast to the original GPU-WAH design, which assumes the input is already a raw bitmap, a dedicated parallel kernel was added in this study. This kernel aims to convert unique integer sets into bitmap format using atomic OR operations on the GPU. This conversion step is performed entirely on-device and enables a fair comparison with HiPES, which processes the integer set directly without any bitmap materialization. The measurements include the execution time of this kernel. Further optimizations are beyond the scope of this study. The CUDA implementation includes adaptations for architectural efficiency. Each phase of the WAH compression process was implemented as a separate kernel. This provides high levels of parallelism, while minimizes thread divergence and facilitates the use of coalesced memory accesses. The correctness was verified by a CPU-based decompression routine followed by integer set reconstruction and comparison with the input. Furthermore, it was cross-validated against an existing open-source version[3]. For more details, please refer to the Artifacts.

4.2 Datasets

Synthetic datasets were generated following the established methodology of [3], a recent peer-reviewed study on large-scale GPU set intersection techniques. This methodology is widely adopted in the recent GPU set-processing literature since it creates reproducible stress cases for parallelism, coalesced accesses, and internal fragmentation, the phenomena that HiPES aims to expose and optimize. Eight experimental scenarios were designed, covering a range of data distributions and densities. The distribution model, the cardinality of the complete universe (e) and the total number of elements (n) were considered to build the datasets. These parameters determine memory-access patterns and thread divergence on GPUs.

The resulting global density (d_g) was determined by the ratio of n/e. The e values of 10^8 and 10^9 were selected. Therefore, it was possible to assess the efficiency and robustness across a broad range of data, from sparse (0.1%) to dense (10%) and using uniform and clustered distributions (Tab. 1). These densities directly correspond to realistic use cases in GNN adjacency-set processing, RAG candidate filtering, and in-situ scientific metadata analytics [27,25,30]. The

[3] `https://github.com/holgus103/GPU-WAH`

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

script to generate the datasets was implemented in C++ using Mersenne Twister pseudo-random number generator to guarantee statistical reproducibility and is available in the Artifacts. Scenarios S1–S4 use a uniform distribution to assess raw processing speed in the absence of data locality, which directly stresses HiPES fixed-layout and its absence of RLE optimizations. Scenarios S5–S8 employ a clustered distribution, which aims to capture element locality. In this way, it is possible to assess memory-level parallelism due to spatial distribution effects.

Tab. 1: Experimental configurations for datasets.

Scenario	Distribution Model	Universe of Elements (e)	Set Size (n)	Proportion n/e
S1	Uniform	10^8	10^6	1%
S2	Uniform	10^8	10^7	10%
S3	Uniform	10^9	10^6	0.1%
S4	Uniform	10^9	10^7	1%
S5	Clustered	10^8	10^6	1%
S6	Clustered	10^8	10^7	10%
S7	Clustered	10^9	10^6	0.1%
S8	Clustered	10^9	10^7	1%

4.3 Hardware, Conditions, and Measurements

An NVIDIA GeForce RTX 2060 GPU (1920 CUDA cores and 6 GB VRAM) was used along with an Intel Core i7 processor and 16 GB RAM. Each configuration had 100 independent runs on an idle machine so that means and standard deviations could be calculated. All algorithms were implemented in CUDA [24] and then compiled using `-O3`. The experiments were run with GPU Boost disabled. The Compression Speed and Ratio are computed using the following equations: $C_{\text{speed}} = (S_i/1024^2)/T_s$ in MB/s and $C_{ratio} = S_i/S_c$, where S_i represents the size of the initial uncompressed input data, S_c represents the size of the final compressed output, and T_s represents the total compression time.

4.4 Study of Parameter w

A key design parameter of HiPES is the parameter w. We evaluate w values from 7 to 29 to investigate its memory-bound behavior. This range can be divided into three main regimes: the first includes $w < 14$. At this range, the small payloads (< 2 KB) produce numerous chunks, resulting in high contention among threads that are performing global atomic operations during the initial compression passes, the second considers $14 \leq w \leq 18$, where payloads (2-32 KB) fit within the 48 KB SMEM limit for on-chip aggregation, and the third covers $w \geq 19$, where large payloads (≥ 64 KB) result in just a few chunks being created. It is important to note that since HiPES uses fixed-size chunk payloads, memory usage is directly reflected by the C_{ratio}. As such, this measures

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

the effects of internal fragmentation due to changes in the value of w. All data transfer time (including from/to host) were accounted for within the measurements to accurately represent overall performance [4,29]. While this method may add overhead to smaller data sets or shorter running kernels it is amortized over larger data sets and has equal impact on both configurations.

5 Results and Discussion

5.1 Compression Speed and Ratio

Our initial analysis fixes the payload size to 8 KB ($w = 16$), the default implementation. HiPES achieves competitive C_{ratio} in dense workloads (S2 and S6) as can be seen in Tab. 2.

Tab. 2: Comparison of Compression Speed (C_{speed}) and Compression Ratio (C_{ratio}) between GPU-WAH and HiPES for a fixed payload size of 8 KB ($w = 16$).

Scenario	**GPU-WAH**		**HiPES**	
	C_{speed} **(MB/s)**	C_{ratio}	C_{speed} **(MB/s)**	C_{ratio}
S1	142.07±2.20	0.67	1714.24±25.37	0.320
S2	**951.47±9.93**	**3.10**	**7761.41±72.90**	**3.20**
S3	16.97±0.27	0.52	753.75±4.79	0.03
S4	157.05±2.22	0.67	4812.82±44.62	0.32
S5	151.93±2.61	1.33	1710.11±26.61	0.32
S6	**1012.51±11.41**	**5.09**	**7797.72±118.03**	**3.20**
S7	19.10±0.19	0.59	751.66±7.90	0.03
S8	172.29±1.18	1.34	4689.17±41.61	0.32

[a] $C_{\text{ratio}} < 1$ indicates data expansion. [b] Scenarios 2 and 6 are highlighted in **bold** since both achieved data compression.

For S2, HiPES performs slightly better given that algorithms based on WAH rely on RLE, which leads to an increased number of literal words [1]. In contrast, HiPES employs a fixed-layout bitmap representation rather than exploiting data regularities. With respect to C_{speed}, HiPES exhibits improvements up to 8 × in dense workloads. This can be attributed to the architectural features discussed in Section 3, which facilitate coalesced memory access, minimize thread divergence, and process input data in SMEM. However, with regard to sparse cluster-based scenarios (S5 and S8), we find a C_{ratio} near 1 for GPU-WAH, since it uses RLE to represent gaps between clusters [29]. Therefore, this ability is not available in HiPES. For S1, S3, S4, and S7, both HiPES and GPU-WAH demonstrate expansion due to high degree of entropy. Entropy represents values that exhibit a large degree of spatial and temporal distribution, thereby hindering bitmap compression. These results illustrate one of the primary constraints of HiPES design, which is its fixed layout structure. It may lead to internal fragmentation when dealing with very dispersed data with low spatial locality. This factor constrains the amount of data that can be stored per chunk for the representation of a few set bits. In terms of C_{speed}, HiPES shows better performance on all tested scenarios. This comparison is especially important since, although both

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

schemes present expansion in sparse workloads, HiPES still offers greater speed than GPU-WAH.

5.2 Effect of the Parameter w

We studied how w affects C_{speed} and C_{ratio} in HiPES by analyzing its alignment with the GPU memory hierarchy. The analysis was limited to HiPES since the w is not available for GPU-WAH. The trend for throughput across all test cases is displayed in Fig. 2 and it follows a bell-shaped curve. For small payloads ($w < 14$), throughput is limited, remaining below ∼5,000 MB/s. This is due to the large number of chunks for the GPU to schedule, forcing the use of GMEM atomics, since metadata will be too large to be cached efficiently. The highest throughput of approximately 7,678-8,129 MB/s occurs when the payload fits into the SMEM ($14 \leq w \leq 18$). Since atomic updates are made directly from this buffer, the performance for both uniform (S2) and clustered (S6) dense test cases were equivalent. For higher payloads greater than SMEM ($w \geq 19$), there is a divergence in throughput. Interestingly, in dense cases, throughput for uniform data (S2) continues to be very fast (≥ 8,000 MB/s) until $w = 21$. This can be attributed to an architectural effect within the GPU related to L2 cache parallelism [15], but it is effectively being utilized by HiPES because of its fixed layout design and regular memory access patterns compared to the random memory access patterns produced by GPU-ported schemes. On the other hand, clustered data (S6) drops to ∼6,000 MB/s as dense clusters target identical addresses, causing serialization bottlenecks. Finally, sparse scenarios (e.g., S3, S7) are limited by metadata overhead (peaking around 1,100 MB/s). However, larger sparse datasets (S4, S8) are nearly 3× faster than smaller ones. This supports the observation that high GPU occupancy is critical to saturate compute resources when data is sparse.

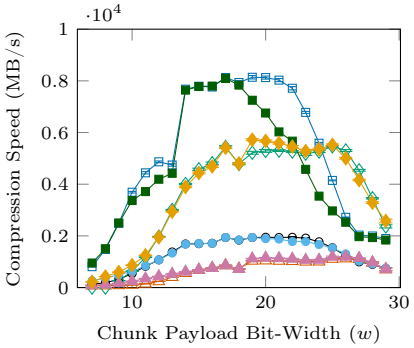


Fig. 2: Compression speed C_{speed} as a function of w across data distributions.

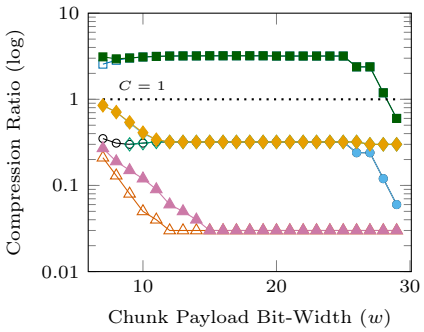
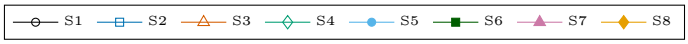


Fig. 3: Compression ratio C_{ratio} as a function of w across data distributions.



ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

While throughput increases with w, the C_{ratio} is determined by data density (Fig. 3), as a result of the HiPES fixed-layout design. In high-density workloads (S2, S6) the C_{ratio} reaches plateau at ≈ 3.2 ($15 \leq w \leq 25$). At that point, the internal fragmentation is minimized because w is aligned to the dense data distribution. However, once $w > 25$ the C_{ratio} will decrease sharply. The reason is that the payload size is now larger, and so, even a single present value will require mostly empty bitmaps to be stored. Therefore, even in high-density workloads, severe internal fragmentation is present. In general, the C_{ratio} for all the other scenarios (S1, S3–S5, S7, and S8) will be less than 1.00, thus showing data expansion. It is worth noting that for small w (< 10), HiPES has a limited mitigating effect for mixed workloads (for example, S5 is only at 0.85 at $w = 7$). Therefore, as w continues to increase, the C_{ratio} will continue to decay until it is near zero. Overall, the sparse workload exposes a fundamental constraint of HiPES fixed-layout, and therefore an adaptive form (like list-based container [16]) will be a better representation than the fixed-size bitmaps.

Additionally, the extent of how HiPES generalizes across hardware architectures is demonstrated. For it, we evaluate the C_{speed} within six different NVIDIA GPU hardware with varying compute capabilities (CC), from Kepler (CC 3.5) to Ada Lovelace (CC 8.9), using the scenarios S2 and S6. It is observed that HiPES throughput scales positively with increasing CC ($r^2 \approx 0.78$). At Ada Lovelace, HiPES achieves ≈12,234 MB/s compared to ≈1,075 MB/s for GPU-WAH (∼11× speedup). These findings confirm the benefits of GPU-native design with new hardware generations. As expected, C_{ratio} is independent of the architecture, relying on data distribution. A detailed description of methodological aspects of this analysis are available in the Artifacts.

6 Conclusions

HiPES improves compression throughput by 8× compared to GPU-WAH in dense workloads. Therefore, it introduces internal fragmentation in sparse scenarios. Considering the evaluated GPU architecture, a payload size of $w = 16$ offers the optimal balance between utilizing SMEM and minimizing this fragmentation. In its current format, HiPES is best suited for dense and spatially clustered applications (i.e., GNN adjacency-set processing, RAG candidate filtering, and in-situ scientific metadata analytics), where local chunk density exceeds the 3.13% breakeven threshold ($n/n_c > 2,049$). Therefore, HiPES is preferred when data locality is guaranteed or when high throughput on dense operations is the primary goal. Conversely, in sparse workloads where $C_{\text{ratio}} < 1$, adaptive formats like Roaring bitmaps remain the primary choice.

This study has two main limitations. First, the study of w parameter was conducted exclusively on the RTX 2060. However, the cross-architecture results above confirm that HiPES throughput scales positively with compute capability, supporting the expected generalizability of the SMEM alignment findings. Second, both HiPES and GPU-WAH were implemented by the authors. However, the GPU-WAH implementation was cross-validated against a publicly available

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

version. It supports the correctness and fairness of the comparison. Future work should investigate methods to overcome the limitations presented above and include validation of the scheme on later-generation hardware and adaptive GPU bitmap formats. Additionally, we plan to evaluate fully GPU-resident query operations that leverage the absence of decompression and move beyond traditional database systems.

Acknowledgments. The authors thank the Federal University of Paraná for support in the development of this research. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Reproducibility. The source code, datasets, and cross-architecture benchmark data are available at `https://github.com/josepedro/hipes`.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Andrzejewski, D., Wrembel, R.: Gpu-wah: Applying gpus to compressing bitmap indexes with word-aligned hybrid. In: 12th International Conference on Data Warehousing and Knowledge Discovery. pp. 150–163. Springer (2010)
2. Andrzejewski, D., Wrembel, R.: Gpu-plwah: Gpu-based implementation of the plwah algorithm for compressing bitmaps. In: 15th East European Conference on Advances in Databases and Information Systems. pp. 97–110. Springer (2011)
3. Bellas, C., Gounaris, A.: An evaluation of large set intersection techniques on gpus. In: 23rd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data. Nicosia, Cyprus, March 23 (2021)
4. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps. Software: Practice and Experience **46**(5), 709–719 (2016)
5. Chan, C.Y., Ioannidis, Y.E.: Bitmap index design and evaluation. ACM SIGMOD Record **27**(2), 355–366 (1998)
6. Chantrapornchai, C.: Combined bitmap representation and its applications to query processing of resource description framework on gpu. In: 2019 International Conference on High Performance Computing & Simulation (HPCS). pp. 720–727. IEEE (2019)
7. Colantonio, A., Di Pietro, R.: Concise: Compressed 'n' composable integer set. Information Processing Letters **110**(16), 644–650 (July 2010)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
9. Corporation, N.: cuCollections: Gpu-accelerated concurrent data structures. `https://github.com/NVIDIA/cuCollections` (2026)
10. Elbek, D., Kaya, K.: Blest: Blazingly efficient bfs using tensor cores. arXiv (2025)
11. Hijma, P., Heldens, S., Sclocco, A., Van Werkhoven, B., Bal, H.E.: Optimization techniques for gpu programming. ACM Computing Surveys **55**(11), 1–81 (2023)
12. Janssen, D., Pullan, W., Liew, A.W.C.: GPU Based Differential Evolution: New Insights and Comparative Study. arXiv (2024)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39

13. Kaser, O., Lemire, D., Aouiche, K.: Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes. In: 11th International Workshop on Data Warehousing and OLAP. pp. 33–40. ACM (Sep 2008)
14. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3. Addison-Wesley Professional, 2nd edn. (1998)
15. Lal, S., Varma, B.S., Juurlink, B.: A quantitative study of locality in gpu caches for memory-divergent workloads. International journal of parallel programming **50**(2), 189–216 (2022)
16. Lemire, D., Kaser, O., Kurz, N., Deri, L., O'Hara, C., Saint-Jacques, F., Ssi-Yan-Kai, G.: Roaring bitmaps: Implementation of an optimized software library. Software: Practice and Experience **48**(4), 867–895 (2018)
17. Lemire, D., Ssi-Yan-Kai, G., Kaser, O.: Consistently faster and smaller compressed bitmaps with roaring. Software: Practice and Experience **46**(11), 1547–1569 (2016)
18. Li, Z., Wang, W., Li, R., Chen, C., Long, X., Zheng, L., Xu, Q., Yang, C.: A high-throughput gpu framework for adaptive lossless compression of floating-point data. arXiv (2025)
19. Mallia, A., Siedlaczek, M., Suel, T., Zahran, M.: Gpu-accelerated decoding of integer lists. In: Proceedings of the 28th ACM CIKM. pp. 2453–2456 (2019)
20. Nelson, J., Xie, T., Sun, Y.: Gpu acceleration of range queries over large data sets. Journal of Parallel and Distributed Computing **125**, 137–149 (2019)
21. Nelson, J., Xie, T., Sun, Y.: Parallel acceleration of cpu and gpu range queries over large data sets. Future Generation Computer Systems **102**, 360–374 (2020)
22. Nelson, J., Xie, T., Sun, Y.: A statistical performance analysis of gpu wah range querying. Concurrency and Computation: Practice and Experience **34**(8), e6754 (2022)
23. Nicholson, H., Chasialis, K., Boffa, A., Ailamaki, A.: The effectiveness of compression for gpu-accelerated queries on out-of-memory datasets. In: 21st International Workshop on Data Management on New Hardware. DaMoN '25, Association for Computing Machinery (2025)
24. NVIDIA: CUDA C Programming Guide. NVIDIA Corporation (2025), release 13.0. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`
25. Pan, J.J., Wang, J., Li, G.: Survey of vector database management systems. The VLDB Journal **33**(5), 1591–1615 (Jul 2024)
26. Silvestri, C., Catarci, T., De Rosa, M., Santucci, G.: Gpu-based computing of repeated range queries over moving objects. Journal of Visual Languages & Computing **25**(4), 494–502 (2014)
27. Tawfik Ibrahim, A.S., Paolini, E., Cugini, F., Paolucci, F.: From packets to predictions on gpu: Accelerated graph-based intrusion detection system. Computer Networks **275**, 111954 (2026)
28. Tran, K., Kim, Y.K.: Exploring means to enhance the efficiency of gpu bitmap index query processing. In: 2021 IEEE International Conference on Big Data and Smart Computing. pp. 53–60 (2021)
29. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems **31**(1), 1–38 (2006)
30. Zhang, B., Davis, P.E., Morales, N., Zhang, Z., Teranishi, K., Parashar, M.: Optimizing data movement for gpu-based in-situ workflow using gpudirect rdma. In: 29th International Conference on Parallel and Distributed Computing. p. 323–338 (2023)
31. Zhang, W., Wu, Z., Wang, W., Zhao, Y.: Hg-bitmap join index: A hybrid gpu/cpu bitmap join index mechanism for olap. In: 2014 IEEE International Conference on Big Data. pp. 32–39. IEEE (2014)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_39